2007

Mobile Multimedia Streaming Library

Bao Ho
San Jose State University

Mobile Multimedia Streaming Library

A Project Report

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the

Requirements for the Degree

Masters of Science

By
Bao Ho

December 2007

Approved by the Department of Computer Science

_____

Dr. Suneuy Kim

_____

Dr. Melody Moh

_____

Dr. Robert Chun

Approved by the University

_____

ABSTRACT

In recent years, multimedia has become a commonly used tool for presenting contents to the users. The employment of multimedia is no longer limited to only the entertainment industry, but spans in other areas as well. In academics, lectures are recorded to audio and video for storage and distribution to students. Free online multimedia hosting services are popularly cherished, such as "youtube.com" and "yahoo video", and with the increasing affordability of digital camera, hundreds, or maybe thousands, of home-made videos and music audio are created daily and published online. Low-cost digital recorders such as webcams also help promote the use of video for surveillance, both for commercial and personal use. Suddenly, there comes the need for digital multimedia delivery, which happens naturally with the advancement in Internet bandwidth and the popularity of multimedia sharing. Multimedia delivery comes in two methods: downloading and streaming. Streaming requires more complex structure, but rewards with better user experience. Although streaming is the method of choice today, downloading is still useful in ad-hoc situation where streaming is not feasible.

This project aims to provide streaming-like capability to mobile devices. Since mobile gadgets are limited in resources compared to personal computers (PC), streaming sometimes is the only way to deliver media contents to user. This work targets devices in the so-called "ad-hoc situation", and also seeks to save the cost associated with multimedia streaming, which traditionally uses the operator wireless network, by using a LAN-connected proxy and the Bluetooth medium. It is also to serve the educational purpose in learning about multimedia streaming on cellular phones.

This project experiments with several approaches to implement streaming on mobile phones. It discusses each approach in details. Finally, a library and a sample application are implemented to demonstrate the solution.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

For a long time, tapes, compact discs, and digital storage such as hard-drive or memory cards, have been the main forms of multimedia contents distribution. The increasing ease of access to the Internet in the last decade, the general population has widely adopted the Internet as the distribution channel for digital contents, especially multimedia. At the early stage, media download was the only method of data delivery. Point-to-Point software like Napster enabled users to share contents by uploading and downloading the contents from other users' machines. This method serves well for users who were willing to start the download process and view the media at a later time. However, as the quality of digital multimedia improves, the size of the media grows substantially. The Internet, even with the latest advancements in network speed, cannot keep up. A user, wanting to see what is in a video, would have to wait for the entire video to be downloaded before he or she could view it. This long delay is inefficient and degrades user experience. Downloading cannot be a solution for applications with stringent requirements for real-time multimedia delivery.

It soon became clear that a new method needed to be derived to satisfy near real-time multimedia delivery requirement. Faster network is not a complete solution, as advancement in network speed is not as fast as advancement in media data. Media contents have to be divided into independent segments, with each capable of being presented to the user. In other words, the media has to be formatted in small presentable units; and since each unit is small enough, it can be transferred to the user's machine quickly for view. This method in effect produces a continuous stream of viewable video segments, and thus the technical term "streaming" was coined [1].

In the last few years, personal computers are no longer the only form of multimedia player used in streaming. As people spend more time on the road, compact digital gadgets such as mobile phone have become popular. The demand for multimedia playback on these small devices also grows. Users want to have the ability to listen to music or watch movies on their mobile phones while waiting at the train station, on bus, etc., and to be able to monitor their house from the remote web camera. The real-time

requirement, the slow cellular network, and the limited resources on mobile devices again confirm the applicability and usefulness of multimedia streaming [2, 3].

This project seeks to deliver multimedia to the mobile device, targeting multimedia-ready devices lacking streaming support. The required configuration is that some multimedia is ready on a remote server, and that the mobile device is capable of interpreting this media data format. Two different approaches have been taken and implemented before the final solution is derived. The first approach uses two network connections, one TCP and one UDP, to handle the communication with the Darwin streaming server. However, the cellular operator only allows TCP traffic while blocking UDP, for security reason, and thus media data cannot reach the device. The second method implements the TCP-interleaving method which uses only one TCP connection. In this case, the mobile device, although able to receive media data, fails to present it because the device is incapable of playing partially streamed media.

It becomes apparent that the mobile device can only play complete media data. Hence, a solution is derived by splitting the original media into multiple sub-clips, which are then downloaded in advance to provide a continuous playback effect. TCP/IP is used for the download. However, TCP/IP is costly on cellular network, and thus Bluetooth is also provided for sub-clip delivery. This multi-subclip solution requires a custom media server, a custom streaming protocol, a Bluetooth proxy, and a client library capable of handling the streaming and Bluetooth protocols. With this approach, all the sub-clips comprising the original media data can be fetched and played successfully.

The paper first gives an overview of multimedia and multimedia encoding in Section 2. It then goes briefly over the two methods of multimedia delivery, downloading and streaming, with the pros and cons, in Section 3. Section 4 describes the standard real-time media streaming in more details, as well as the common protocol implementations. Section 5 discusses multimedia streaming on mobile devices. The design of the mobile streaming library – the goal of this project – is covered in Section 6. We conclude the project in Section 7, and provide some potential future work in Section 8. The material used as references in this project is listed in Section 9.

2 DIGITAL MULTIMEDIA BASICS

Multimedia production begins with the recording process, in which the moving picture and possibly sound are captured using a camera. If an analog camera is used, as in the early days, the resulting analog media data has to be digitized. The raw digital data is then processed - encoded to a specific format, compressed, and stored in a container file – for the media player to interpret and present on the intended display.

Multimedia contents contain one or more channels of information, also called tracks, consisting of text, graphics, animation, video, and audio. Each track delivers a different type of information, i.e. video and audio deliver the media contents, while text provides subtitle/translation and graphics delivers interactivity to enhance user perception.

A digital audio or video track is a continuous sequence of still sound or pictures. Each of these still units is a frame, a snapshot of the media a single point in time. The frames are captured and displayed at constant rate, called frame per second (fps). When played one after another, the frames seem to be moving. When played at high speed, faster than the human eyes can differentiate, we perceive a moving audio or video.

Considering a video frame of 720 pixels wide by 480 pixels high, each uses a color depth of 24 bits (3 bytes), will need 720x480x3 or 1 megabyte (MB) of storage. If the frames are captured at 25 fps, it takes 25 MB per second, or 1.5 gigabyte (GB) per minute, or 90 GB per hour of movie. The higher quality or bigger dimension, the longer it takes to digitize and convert the video to a suitable format for display. It is impractical to support video of big sizes and high quality, either for storage or distribution. Thus, multimedia production and usage is an expensive and time-consuming process.

2.1 ENCODING/DECODING

When moving pictures or audio signals are captured and digitized, they are usually in the raw format, and need to be converted to a format suitable to a particular medium type, i.e. MPEG2 format for video on DVD, MP3 format for audio, etc. This process is called encoding. The media format specifies how the media data is structured, how it should be delivered to the display device, and how it should be interpreted. Since moving pictures

and sound are represented as frames, the format describes the number of frames per second, the structure of the frames, the relation between consecutives frames, how each frame is represented, how frames from different tracks, such as audio and video in a movie, are to by synchronized in time, etc. In contrast to encoding, which is the media producer, and may be done directly by a digital recorder or separately on the complete media contents using transcoding software or hardware, decoding is used on the consumer side. The media player must be capable of understanding the format of the encoded media to decode and present it on a television or computer monitor. The decoder reads in the data streaming and divides the data into frames. In some format, frames are not independent, and must be recreated from previous and/or following frames, as in MPEG4. Then, the frames can be displayed on the display device.

2.2 COMPRESSION

In contrast with television, which has the fully dedicated cable infrastructure for data transfer, multimedia over the Internet is a disappointment. Due to bandwidth restriction of the telephone system, the most popular method of network multimedia delivery, it is just not possible to deliver and display full-motion video with stereo sound. Low-quality, or low-bandwidth, media is not appealing to the viewers, and thus media data shared over the Internet are usually limited to short and small dimension videos.

The answer to the media giant delivered over the small-pipe network is compression. Media compression refers to the process of transforming the data to use fewer bits. Before multimedia can be efficiently compressed, media contents must be filtered to keep only the necessary information and throw away redundant data – information that does not contribute to the user perception of the media. Psycho-acoustical research teaches us that there are certain sound frequencies and color spectra that the human ears and eyes cannot detect or tell whether they are included or not. Thus, this useless information can be safely thrown away without affecting the perceived data. Also, the human eyes cannot distinguish small differences in color, and thus groups of very similar colors can be averaged out or generalized in bigger groups. Other redundant information such as the black background enclosing the viewable area can also be filtered to further cut down media size.

Media compression techniques are categorized as either lossless or lossy. With lossless techniques, some considerable level of data compression can be achieved while guaranteeing full reconstruction of the original data. Lossless compression algorithms, while preserving the media data, seek to represent the same data using as few bits as possible. This process employs tricks such as writing consecutive and similar data using shorter syntax, i.e. "BLUEx50" to represent fifty consecutive words "BLUE", using shorter code to represent the most frequently occurring binary data (as used in Huffman coding), writing only the difference between consecutive frames, or using a color map and storing only the index to the color in the map instead of the longer full color code.

Lossy compression yields very high compression rate at the cost of degrading media quality. This process throws away data, redundant or not, at each level of compression. This includes using lower bit-rate, lower sampling rate and/or frame rate, reducing video dimension or audio volume, and shorter media duration, etc. Using lossy compression methods, the media data can be compressed significantly, i.e., by shrinking to half of its original width and height, the size is reduced by a factor of four. Lossy compression is an irreversible process; and the higher the compression rate is used, the lower the media quality becomes.

2.3 MEDIA CONTAINER

Media data are encoded, compressed, and contained in a computer file called a container. A container is used to interleave, or mix, different data types in a specified format, allowing the data to be retrieved in such a way that is most suitable to the data consumer. Video data usually contain video, audio, and optionally chapter and subtitle tracks. These components are synchronized in time, and must be retrieved, decoded, and displayed at the same time. Containers allow multiple tracks to be interleaved in one or multiple files, and retrieved for playback in synchronized manner, as if each frame represents data for all the components. Container files also carry meta-data (tags) besides media data. This meta-data describes the different components in the container, as well as information required for stream synchronization. There are many container formats: WAV, AIFF, AVI, ASF, MOV, OGM, MP4, 3GP, etc. Among these, 3GP is designed for mobile devices, and thus is used in this project.

2.4 MEDIA HINTING

Before a file can be streamed – being retrieved and viewed at the same time – it must be hinted. Hinting adds information about the various tracks in a media file that tells a streaming server how to read and serve the data to a streaming client. Some encoding software, such as QuickTime, also hints the converted media file. Hinting can also be done on the complete converted file using software such as the open source GPAC tool suite.

Below is a video file that has not been hinted. *Mp4info* tells us that there are two tracks in the video: a video track encoded in MPEG-4 Simple format, and an audio track encoded in MPEG-4 AAC format.

```
C:\ >mp4info UnhintedKRON.3gp

mp4info version 1.4.15
UnhintedKRON.3gp:
Track    Type     Info
1        video    MPEG-4 Simple @ L0, 59.750 secs, 45 kbps, 176x144 @ 8.000000 fps
3        audio    MPEG-4 AAC LC, 59.648 secs, 12 kbps, 8000 Hz
```

Figure 1. Un-hinted media file

The following is the content description of the same video file, but is hinted. There are two additional tracks describing the two main tracks. These latter tracks are used as clues to the streaming software to stream the contents to the client.

```
C:\ >mp4info HintedKRON.3gp

mp4info version 1.4.15
HintedKRON.3gp:
Track    Type     Info
1        video    MPEG-4 Simple @ L0, 59.750 secs, 45 kbps, 176x144 @ 8.000000 fps
3        audio    MPEG-4 AAC LC, 59.648 secs, 12 kbps, 8000 Hz
65536    hint     Payload (null) for track 1
65538    hint     Payload (null) for track 3
```

Figure 2. Hinted media file

3 MULTIMEDIA RETRIEVAL METHODS

Besides the many ways of delivering a movie to a user via physical media: on a VHS tape, a CD or DVD, etc., there are only two ways of delivering digital audio/video over the Internet: download or streaming [17].

3.1 DOWNLOAD

Downloading requires the user to wait for the download process to complete before he or she can start viewing. Throughout the download delay, the partial data is unusable. The download process is also more susceptible to failure, as many things can go wrong, such as a broken connection, errors on the server side or client side, user losing patience and canceling the download, etc.

Once a media is completely downloaded, it is literally guaranteed to be without glitches. It can be stored on local storage and played back as many times as the user desires. The media plays smoothly without any delay besides the limitations of the hardware. However, response time plays a major role to user experience. With millions of video clips on the Internet, most of the time a user will be skimming through the first few seconds of the video before deciding to view the entire clip. Downloading does not facilitate that experience. The user may waste time and bandwidth downloading something he or she may not like, or most likely will skip that video. It is said that the easiest way to discourage the audience or kill a movie is a big fat download delay.

3.2 STREAMING

Streaming allows the user to view the media while it is still downloading. After starting the streaming process, the user waits for some initial data to be delivered. As soon as enough data is on the client side, it is played, while more data is still being downloaded. This process continues incrementally until the end. In its simplest form, streaming works like a pipe: buffered data is played on one end, while new data comes in on the other end to fill up the pipe.

There are many benefits to streaming, such as shorter wait time, preview feature, more tolerant of failure, less memory requirement, and real-time playback. Although there still is a delay at the beginning, it is insignificant compared to media download.

The user gets to see, or preview, what is in the movie before he or she is committed to getting the entire file. If some data is corrupted in the middle of the streaming session, the streaming software can throw away that unusable data, and waits until good frames arrive. Although the user will see a gap, it is still far better than having to restart the entire movie, as with downloading.

Memory footprint is the amount of memory required for an application to operate, i.e., the memory required to open QuickTime and play a movie. It is most likely not a problem with a personal computer, but is an important factor that decides the success of multimedia on small devices, such as mobile phone. These gadgets have little memory, with a few megabytes at the low end. It is not acceptable, if not impossible, to download the entire video onto these devices for playback. High data transfer cost on cellular networks greatly helps user learn to appreciate the preview feature.

For real-time applications such as video surveillance, there is no concept of media start or stop time. The video starts when the user wants to initiate the monitoring process, and continues indefinitely until the user stops it. In this case, media download is inapplicable, and streaming is the only option.

Multimedia streaming, although offering many benefits, carries quite a few drawbacks. For example, when network is slow, data cannot be delivered as fast as it is consumed, the user will experience more "buffering" delays in between. Since user experience must be honored as much as possible, the video data rate has to well match the playback rate. Regardless of how much more network speed may be improved, there is always a lot more data than the network can carry, and network capacity will always be the bottleneck for data transfer. This limits streaming to low bit-rate, low frame-rate, and small dimension multimedia contents. Thus, streaming is most useful to people who want to quickly view the media in exchange for quality, while downloading is for the patient.

4 REAL-TIME MEDIA STREAMING

4.1 PROTOCOL DESCRIPTION

Real-time media streaming is a specification consisting of these standard protocols: Session Description Protocol (SDP), Real-time Streaming Protocol (RTSP), Real-time Transport Protocol (RTP), and Real-time Control Protocol (RTCP).

4.1.1 SESSION DESCRIPTION PROTOCOL (SDP)

Session Description Protocol (RFC 4566) [4] describes the multimedia contents, and is used to deliver meta-information about the media to the client. The client uses this information to negotiate and establish a streaming session with the streaming server. This meta-data includes the following:

- Connection: network type, address type, and address.
- Bandwidth
- Session: session version and id.
- Media properties: media protocol, media type (audio/video), tracks, track ids, track durations, track mapping, dimension and frame size (for video), encoding formats, etc.

Figure 3 shows a video's meta-data in SDP format.

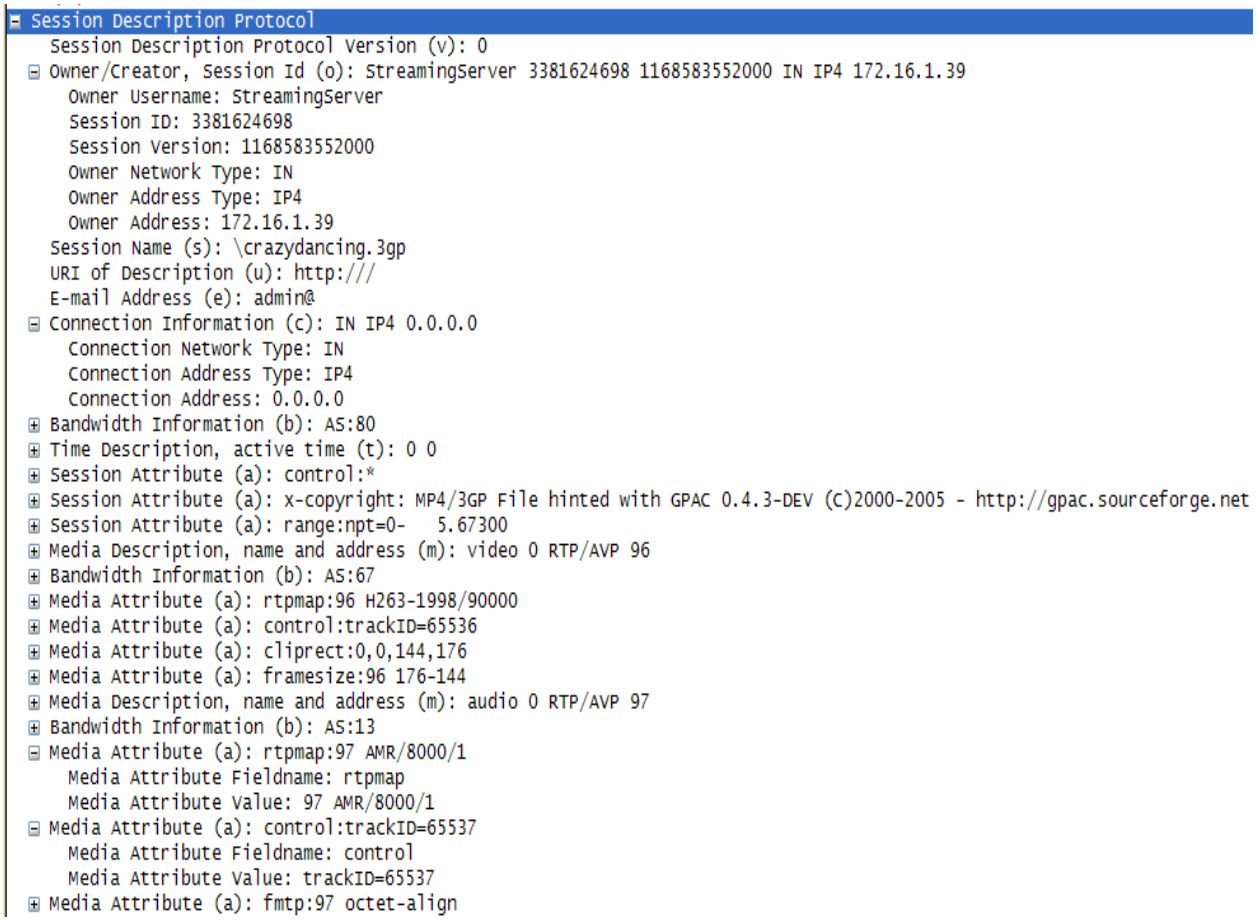


Figure 3. SDP captured using Ethereal

Most important among these properties are the session id, track id, and track type for each track (also called stream). Session id is used in all later requests. Track type is used by media handlers on the client, and track id is used to request a specific track/stream. SDP is used by RTSP at the start of the streaming session.

4.1.2 REAL-TIME STREAMING PROTOCOL (RTSP)

Real-time Streaming Protocol, an IETF standard proposed in RFC 2326 [5], is designed to allow a client to remotely control the streaming session, using VCR-like commands "Play", "Record", "Pause", "Resume", and "Stop". RTSP standardizes the interaction and message exchange between the client and server, and specifies the session life-cycle. The client sends RTSP requests to the server to learn about server capability (OPTIONS)

and media description (DESCRIBE), to establish the session (SETUP), to control the session (PLAY, RECORD, PAUSE) and to terminate the session (TEARDOWN). RTSP request format is similar to that of HTTP. However, unlike HTTP, RTSP is a stateful protocol. RTSP commands and response follow HTTP syntax: each line is terminated with a pair of Carriage-Return/Line-Feed (CRLF), and the last line is a blank, also ending in a CRLF. Both the server and client need to maintain the session state, and transition from one state to the next, or previous, as requested by the RTSP command and response.

A streaming session starts out with the client sending an OPTIONS inquiry about supported operations to the server. The server then responds with the supported operations on that media.

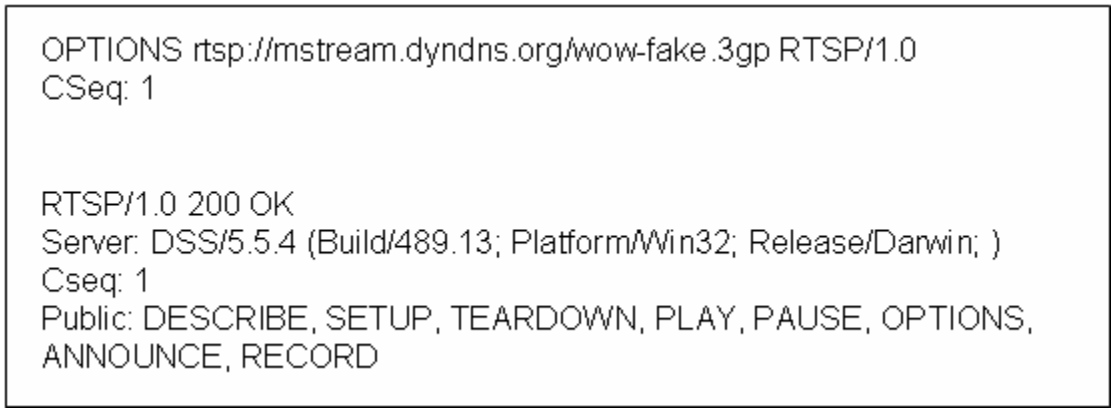


Figure 4. RTSP OPTIONS request and response

Next, the client requests the server for a description of the media, and server sends back the meta-data in SDP format.

```
DESCRIBE rtsp://mstream.dyndns.org:554/crazydancing.3gp RTSP/1.0
CSeq: 2


RTSP/1.0 200 OK
Server: DSS/5.5.4 (Build/489.13; Platform/Win32; Release/Darwin; )
Cseq: 2
Last-Modified: Fri, 12 Jan 2007 06:32:32 GMT
Cache-Control: must-revalidate
Content-length: 519
Date: Wed, 28 Feb 2007 04:11:39 GMT
Expires: Wed, 28 Feb 2007 04:11:39 GMT
Content-Type: application/sdp
x-Accept-Retransmit: our-retransmit
x-Accept-Dynamic-Rate: 1
Content-Base: rtsp://mstream.dyndns.org:554/crazydancing.3gp/

v=0
o=StreamingServer 3381624698 1168583552000 IN IP4 172.16.1.39
s=\crazydancing.3gp
u=http:///
e=admin@
c=IN IP4 0.0.0.0
b=AS:80
t=0 0
a=control:*
a=x-copyright: MP4/3GP File hinted with GPAC 0.4.3-DEV (C)2000-2005 - http://gpac.sourceforge.net
a=range:npt=0-   5.67300
m=video 0 RTP/AVP 96
b=AS:67
a=rtpmap:96 H263-1998/90000
a=control:trackID=65536
a=cliprect:0,0,144,176
a=framesize:96 176-144
m=audio 0 RTP/AVP 97
b=AS:13
a=rtpmap:97 AMR/8000/1
a=control:trackID=65537
a=fmtp:97 octet-align
```

Figure 5. RTSP DESCRIBE request and response

The client establishes the streaming session by sending the SETUP command. The following snapshot shows that the server replies with the transport type (RTP-over-UDP) and the server port the client must connect to for each track (6970 and 6971).

```
SETUP rtsp://mstream.dyndns.org:554/wow-fake.3gp/trackID=65536 RTSP/1.0
CSeq: 3
Transport: RTP/AVP/UDP;unicast;client_port=23996-23997


RTSP/1.0 200 OK
Server: DSS/5.5.4 (Build/489.13; Platform/Win32; Release/Darwin; )
Cseq: 3
Last-Modified: Thu, 22 Mar 2007 05:02:50 GMT
Cache-Control: must-revalidate
Session: 41201621292524
Date: Thu, 05 Apr 2007 05:15:31 GMT
Expires: Thu, 05 Apr 2007 05:15:31 GMT
Transport: RTP/AVP/UDP;unicast;source=172.16.1.41;client_port=23996-23997;server_port=6970-6971;ssrc=0000727F
```

Figure 6. RTSP SETUP request and response

After this, the client can send PLAY to start receiving streamed data. The server starts sending the media data contained in RTP packets following the response.

```
PLAY rtsp://mstream.dyndns.org:554/crazydancing.3gp RTSP/1.0
CSeq: 4
Session: 91061896640550


RTSP/1.0 200 OK
Server: DSS/5.5.4 (Build/489.13; Platform/Win32; Release/Darwin; )
Cseq: 4
Session: 91061896640550
Range: npt=0.00000-5.67300
RTP-Info: url=rtsp://mstream.dyndns.org:554/crazydancing.3gp/trackID=65536;seq=26881;rtptime=3468
```

Figure 7. RTSP PLAY request and response

PAUSE is sent to temporarily stop the streaming.

```
PAUSE rtsp://mstream.dyndns.org:554/wow-fake.3gp RTSP/1.0
CSeq: 5
Session: 41201621292524


RTSP/1.0 200 OK
Server: DSS/5.5.4 (Build/489.13; Platform/Win32; Release/Darwin; )
Cseq: 5
Session: 41201621292524
```

Figure 8. RTSP PAUSE request and response

Finally, to request that the session be terminated, the client sends the TEARDOWN request, as follows:

```
TEARDOWN rtsp://mstream.dyndns.org:554/wow-fake.3gp RTSP/1.0
CSeq: 6
Session: 41201621292524


RTSP/1.0 200 OK
Server: DSS/5.5.4 (Build/489.13; Platform/Win32; Release/Darwin; )
Cseq: 6
Session: 41201621292524
Connection: Close
```

Figure 9. RTSP TEARDOWN request and response

After a session is closed down, any further request sent will be answered with a "Bad request" response.

The following diagram summarizes the RTSP requests and responses involved in a streaming session.

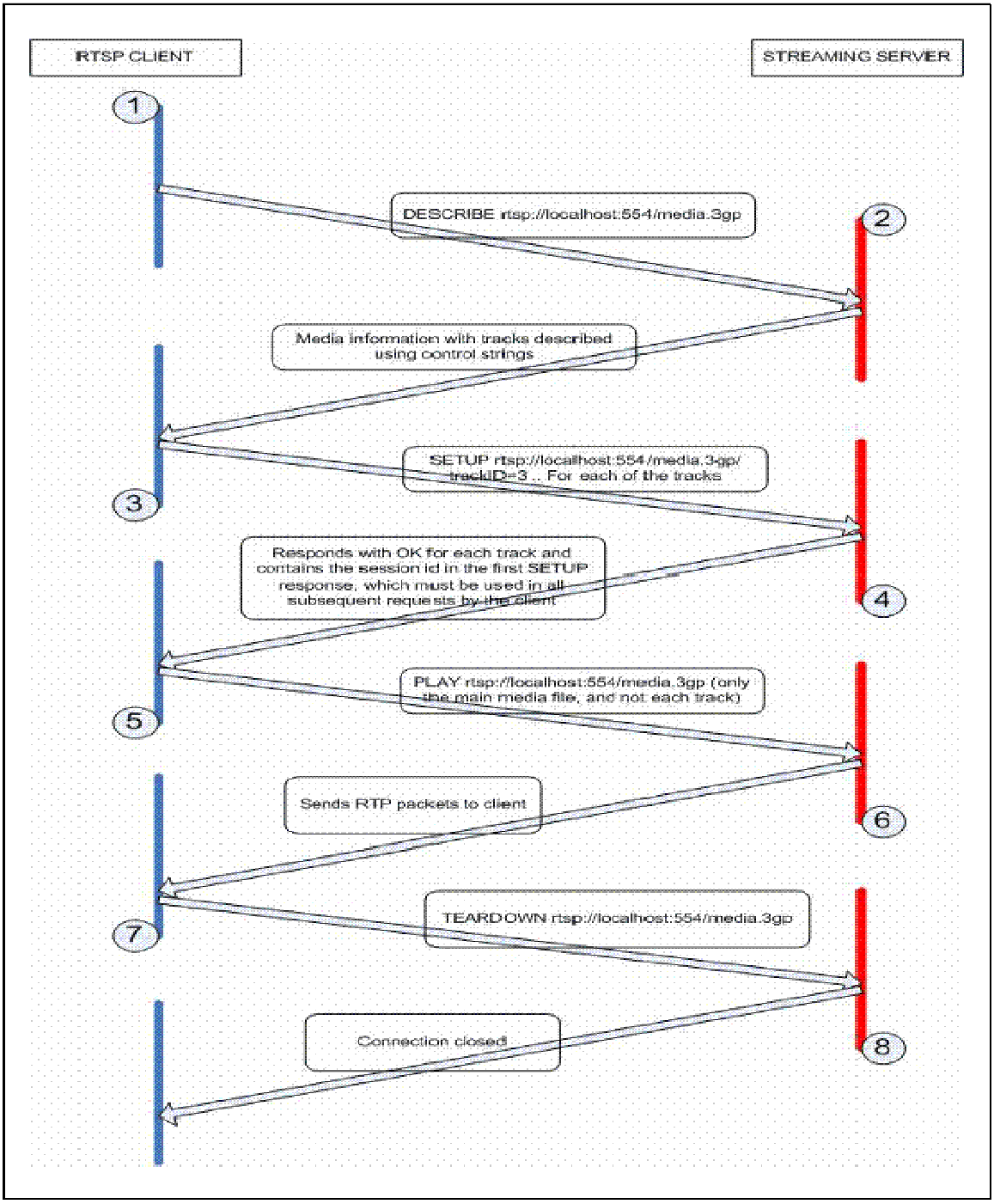


Figure 10. RTSP interaction diagram [10]

4.1.3 REAL-TIME TRANSPORT PROTOCOL (RTP)

Real-time Transport Protocol [6] is an application protocol used for transporting real-time data. It is specified in RFC 3550, and defines end-to-end network transport functions for transmitting data over unicast or multicast network services. RTP does not dictate the underlying network and transport layers, nor does it guarantee quality of service (QoS).

In real-time application, on-time data delivery is more usually important than guaranteed delivery. For example, in a telephone call, we would prefer hearing the other party's voice with as little delay as possible, and would rather repeat the sentence than breakups in the conversation. Since data delivery is not guaranteed, UDP can offer higher speed and does not suffer delay incurred by retransmission. UDP is a better candidate for real-time requirement, and thus is used in RTP implementations to carry RTP data in its payload.

4.1.4 REAL-TIME CONTROL PROTOCOL (RTCP)

Real-time Control Protocol [6] (also defined in RFC 3550) is used to complement RTP. It compensates for the lack of QoS in RTP, by providing out-of-band transmission statistics, control and feedbacks. In a conference telephone call, or in a multimedia session with multiple audiences, RTCP specifies the rate at which the participants, either sender or receiver, can send reports about the RTP packet transmission and reception. Since there is no guarantee that RTP packets will get to the receiver, RTCP Sender Report is used to inform receivers about the transmitted RTP packet count, the sent octet count, the current RTP sequence number, jitter, delay, packet loss, and timestamp used for synchronization. Likewise, RTCP Receiver Report tells the sender about the RTP packet statistics on the receiving side.

RTCP also carries Source-Description (SDES) packet containing information about the participant, such as name, email, phone number, location, etc. RTCP BYE packet is sent when a participant leaves a multi-user session. There is also an Application-defined packet type, intended for experimental use in new applications or features.

RTCP packets are encapsulated in UDP packets, and since RTCP packets are usually small, they can be combined to occupy the entire UDP payload. There are certain

rules for controlling RTCP channel bandwidth allocation. Please consult RFC 3550 for more information about RTCP specification.

4.2 COMMON PROTOCOL IMPLEMENTATIONS

The real-time streaming standard identifies three channels: RTSP, RTP, and RTCP, without mandating how they are implemented. However, the standard specifies the characteristics of the data channels, and provides recommendations for the IP network. RTSP commands and responses require accuracy and guarantee of service. This is best served by TCP/IP since TCP provides retransmission to guarantee that the data will be received and in the correct order. RTCP data is less important and thus can be transmitted over UDP. RTP packets, due to more focus on being on time and less on guarantee of delivery, is also best served by UDP.

Currently, there are three non-proprietary methods of implementing real-time streaming on the IP network. These methods are used for different network configurations: open access to both TCP and UDP, access only to TCP, and only indirect HTTP access via proxy.

4.2.1 RTSP-OVER-TCP / RTP-OVER-UDP [5]

The real-time streaming standard identifies three channels: RTSP, RTP, and RTCP, without mandating how they are implemented. Naturally, and if possible, utilizing UDP for RTP is the better choice. Thus, for networks allowing both TCP and UDP, which is the usual configuration for PC, the RTSP-over-TCP and RTP-over-UDP are used. RTSP requests and responses are not transmitted frequently, not time-critical, require high accuracy and guarantee of service, and thus are transmitted via TCP. Also, in this network configuration, UDP traffic is not fire-walled. Streaming client can send and receive RTP and RTCP packets via UDP. This provides better throughput for real-time data.

This is the ideal method for streaming, where different types of data are communicated based on different requirements to better utilize the network capacity. A firewall can still be employed with special policy to allow UDP traffic without compromising the network security.

The following diagram shows three distinct connections, established to serve three different channels.

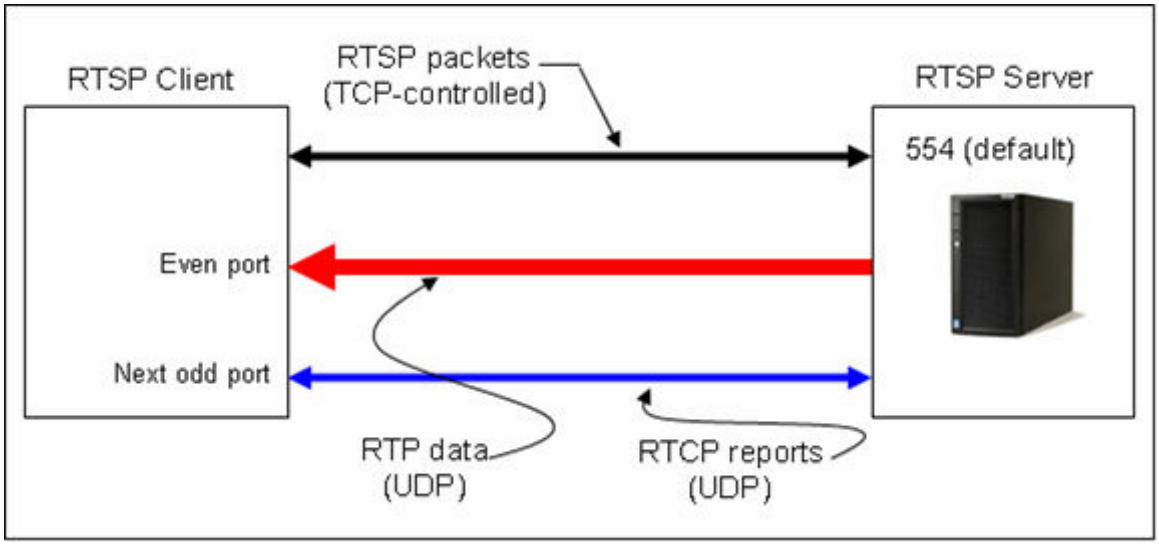


Figure 11. Standard streaming using TCP and UDP

4.2.2 INTERLEAVING-OVER-TCP [5, 7]

Some networks are more restrictive and only allow outbound-established TCP connections. This is most often seen in mobile networks employing the *General Packet Radio Services* (GPRS) system to transmit IP packets. In this situation, UDP cannot be used, and if used, packets will be blocked at the carrier's IP gateway. The RTSP/RTP streaming standard also specifies an alternative method of interleaving. There is only one full-duplex TCP connection, initiated outbound from the client to the external server, and thus can go through the firewall.

Different types of data are communicated over the same TCP connection. However, RTP and RTCP packets must be distinguishable and therefore are delimited by an ASCII dollar sign ($), to indicate the start of the packets. It is followed by a one-byte channel identifier (similar to the port number), a two-byte length, and finally the RTP or RTCP packet.

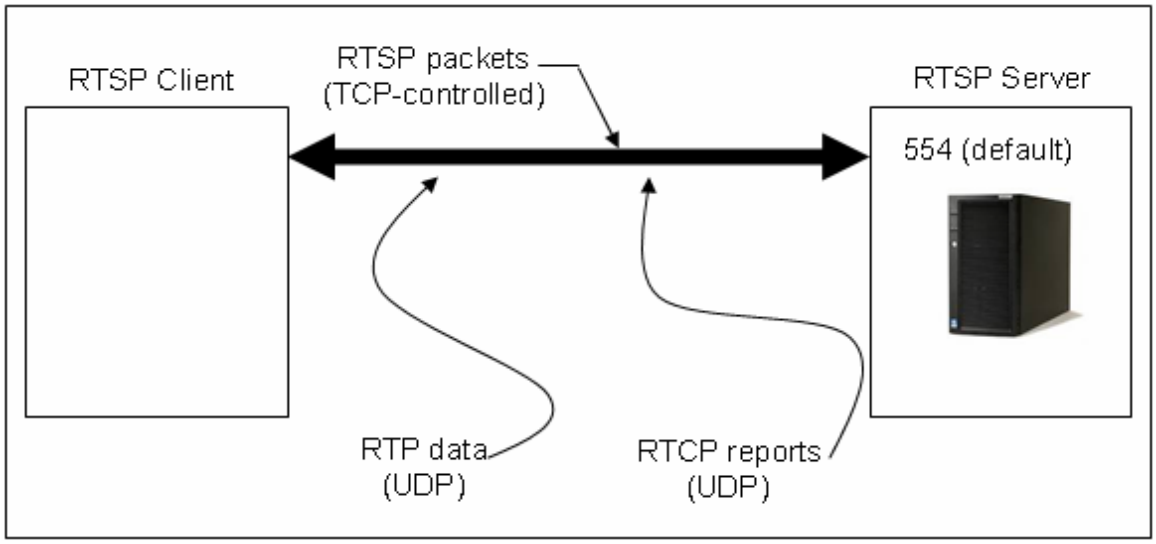


Figure 12. Streaming using TCP-Interleaving

The RTSP commands and responses are similar to the TCP/UDP case. The differences lie in the SETUP command and how the RTP/RTCP data are transferred.

```
SETUP rtsp://mstream.dyndns.org/nicefake.3gp/trackID=65537 RTSP/1.0
CSeq: 16
Transport: RTP/AVP/TCP;unicast;interleaved=2-3
Session: 89648852373023


RTSP/1.0 200 OK
Server: DSS/5.5.4 (Build/489.13; Platform/Win32; Release/Darwin; )
Cseq: 16
Session: 89648852373023
Last-Modified: Thu, 22 Mar 2007 04:00:21 GMT
Cache-Control: must-revalidate
Date: Sun, 25 Mar 2007 21:57:04 GMT
Expires: Sun, 25 Mar 2007 21:57:04 GMT
Transport: RTP/AVP/TCP;unicast;interleaved=2-3;ssrc=00001C49
```

Figure 13. RTSP SETUP request and response using TCP-Interleaving

Here, notice the parameter *interleaved=2-3* in the SETUP request, informing the server to use TCP-interleaving, allocating channel number 2 for RTP and 3 for RTCP. The server agrees and confirms the request of using interleaving.

Next, RTP and RTCP packets are transferred following this format.

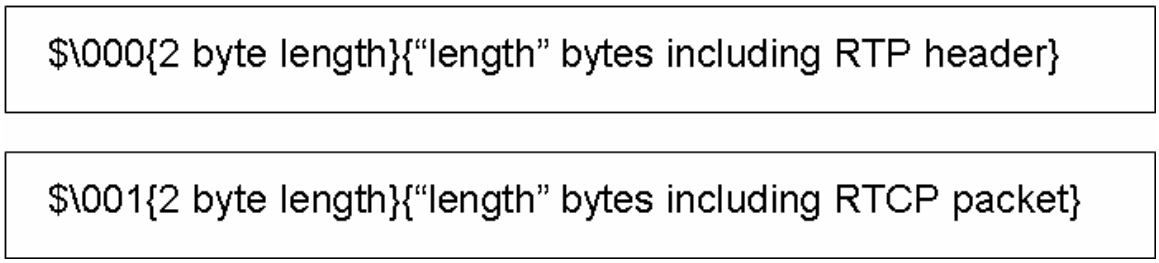


Figure 14. Interleaved RTP and RTCP packet formats

TCP-interleaving by design cannot be as efficient as the TCP/UDP approach because of one single connection versus two dedicated connections. By delivering RTP packets in a connection-oriented fashion, real-time requirements cannot be satisfied. There is also the extra cost to handle the complexity of the interleaving and de-interleaving the data. Thus, unless the network configuration forces us to, TCP/UDP should be used in favor of TCP-interleaving.

4.2.3 TUNNELING-OVER-HTTP

Another variation of TCP-interleaving, called tunneling-over-HTTP [8, 9], is used in the most restrictive networks, where only indirect web-browsing (HTTP using TCP port 80) via HTTP proxy servers is allowed. Since the HTTP proxies run over TCP and only service HTTP clients from the same network, RTSP, RTP and RTCP packets are transmitted over the same TCP connection on port 80, but disguise as HTTP packets. The client originates all the requests, including getting the RTP data. This procedure requires that all requests and replies are not cached by HTTP proxy servers, requests can be identified as pairs to form a full-duplex connection, and that related requests are ensured to connect to the same RTSP server in spite of load-balancing systems using multiple HTTP servers.

Figure 15. Streaming via HTTP-Tunneling [9]

In the above diagram, all RTSP, RTP, and RTCP traffic is done via HTTP GET or POST request/response to convey indefinite amount of data in the reply and message body respectively. In the case of using POST, the RTSP request must be base64-encoded to prevent HTTP proxy server from interpreting the RTSP request in the POST body as a malformed HTTP request.

It should be noted that tunneling-over-HTTP has the worst efficiency and performance. It also suffers the most complexity. However, Quicktime Streaming Server claims it can support this method successfully. See "*Tunneling RTSP and RTP Over HTTP"* for more details.

5 MOBILE STREAMING

Media streaming on mobile device is similar to streaming on a PC. The only differences are in the network media between the carrier and mobile handset, and the media contents. In fact, both PC client and mobile streaming client can use the same server with no change in configuration.

A PC client uses the IP network to access the media content, while a mobile client uses both the mobile wireless GPRS network and the IP network.

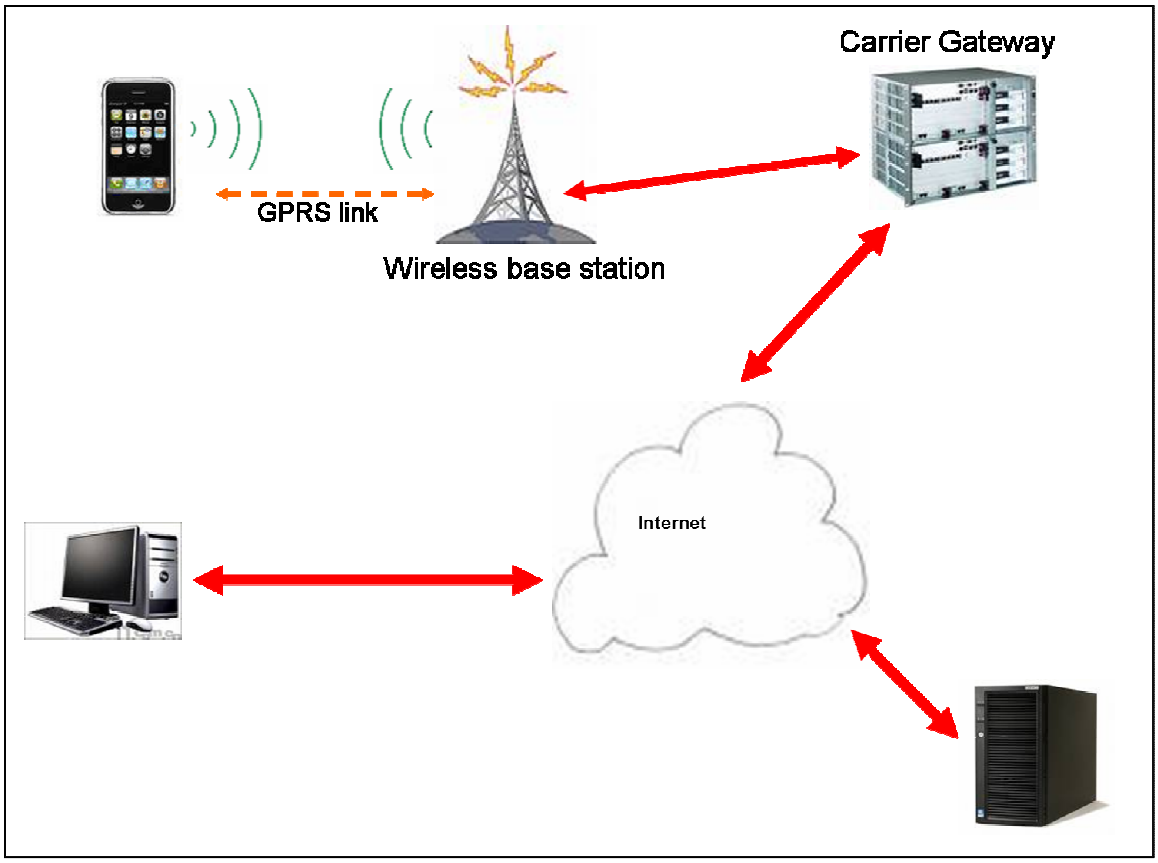


Figure 16. Mobile streaming architecture

As depicted in the picture above, the cellular carrier gateway runs on the IP network, similar to the PC client. The gateway bridges the two network technologies together: the GPRS wireless and IP networks. The gateway converts IP packets to GPRS packet format and then forwards them to the cellular tower, which again forwards to the cellular device.

In terms of media content difference, mobile devices have significantly smaller screen size, compared to PC. Also, the cellular network runs at much slower speed than the IP network. These are two of the several main reasons why streaming of only small dimension, low quality multimedia contents is supported.

To enable multimedia playback hosted at a remote machine, two basic capabilities are needed: 1) network connectivity to establish a connection with the remote server for media delivery, and 2) the ability to play the delivered media. The J2ME Generic Connection Framework (GCF) is the approach to network connectivity, and satisfies the

first requirement. The second requirement is made possible by Multimedia API (MMAPI) that provides multimedia playback. We will discuss GCF and MMAPI in the following sections.

5.1 GENERIC CONNECTION FRAMEWORK (GCF)

J2ME does not provide the full network protocol stack seen on Windows or UNIX systems. Such a full-blown implementation is too resource-intensive for these small-footprint devices. Instead, a simple and lightweight system called Generic Connection Framework (GCF) is used to create the network connection at runtime. When a connection is required, GCF dynamically looks up a class implementing the protocol name specified in the request URL. For example, the following code creates and opens a TCP socket connection to the URL *host.domain.com:port*

(SocketConnection)Connector.open("socket://host.domain.com:port");

and the following code is used to create a UDP connection

(DatagramConnection)Connector.open("datagram://host.domain.com:port");

GCF is the only way to implement network application in J2ME, and will be used to implement media delivery in the streaming library.

5.2 J2ME MULTIMEDIA API (MMAPI) [11]

Multimedia on mobile device running Java is handled by the J2ME Multimedia API (MMAPI) of the JSR 135 specification. It provides a simple and flexible framework for playback and recording of audio and video on resource-constrained devices.

Multimedia processing involves the following steps:

- Protocol handling: retrieves media content from a source such as local storage, database, or streaming server and feeds the content to the media-handing system.
- Media content handling: parses, decodes and renders the media content to output subsystem such as the audio speaker and display screen.

MMAPI defines the high-level interfaces to abstract media retrieval and rendering. Device manufacturers will provide the implementation that best fits their products. MMAPI is intended for powerful devices with advanced multimedia capabilities such as

personal digital assistant (PDA) and the very high-end mobile phones. For the mass-market mobile devices, the Mobile Information Device Profile (MIDP) 2.0 (JSR 118) [19], which is part of the J2ME framework, provides a compatible subset of multimedia functionality. MIDP 2.0 Media API serves as the building block for MMAPI, and is directly compatible. The most important feature that differentiates MIDP Media API from MMAPI is the lack of support for custom DataSource, which is excluded in MIDP 2.0.

5.2.1 PROTOCOL HANDLING

Protocol handling involves streaming session establishment, session management, media request and response, and media delivery. First, a connection must be opened to the remote streaming server. Next, the session is established between the client and the server to prepare for media delivery. This involves information exchanges between the two entities to agree on session parameters. Session states must be maintained on both server and client, and requests and responses are exchanged to let the server know which piece of media the client is interested in. The media contents can then be transported to the client for playback. Client terminates the network connection when the streaming session finishes.

All of the steps above require the client to communicate with the server over the internet. Hence, GCF is used for protocol handling.

5.2.2 MEDIA PLAYBACK

In MMAPI, the interfaces *DataSource* and *Player* are defined for protocol handling and content handling. A DataSource represents the source of the media content. It implements the specific protocol and encapsulates the details of how the media is retrieved using that protocol. Different DataSource implementations are provided to support various sources, such as the *file* protocol, the *http* protocol, and *rtsp* protocol. DataSource has the following main operations:

- connect() – connects to the remote streaming server.
- disconnect() – terminates the session with the streaming server.
- getStreams() – return all the tracks or streams.

- start() – initiates server to start sending data.
- stop() – stops the media data transfer.

The *Player* interface controls the rendering of time-based audio and video contents. It defines methods to manage the player's lifecycle, to create media controls used to manipulate the presentation (audio volume control, video display control, etc.), and to "provide the means to synchronize with other *Players*" [11].

Player has five states in its lifecycle: UNINITIALIZED (player created), REALIZED (media information acquired), PREFETCHED (scarce and exclusive system resources, i.e. audio device, acquired), STARTED (playing), and CLOSED (player destroyed and resources released). Some of the Player's main operations are: realize(), prefetch(), deallocate(), start(), stop(), and close(). The following diagram describes the *Player*'s state machine.

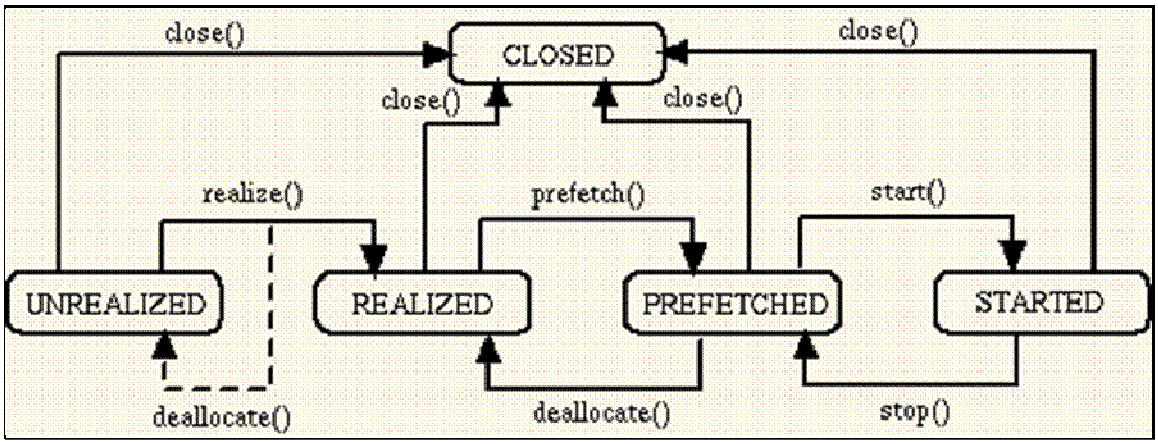


Figure 17. J2ME MMAPI player state machine

The transition operations may in turn invoke operations on the DataSource object to send requests to set up the streaming session, query for media information, start the data transfer, pause the stream(s), and close or tear down the session.

Player instances are created by the factory class *Manager*. This class provides three methods to create players:

1. Using an *InputStream* obtained by reading a file, memory, or network connection.
2. Using a URL location to a remote location.

3. Using a *DataSource*, as mentioned above.

The following diagram shows how these interfaces work together to enable multimedia playback.

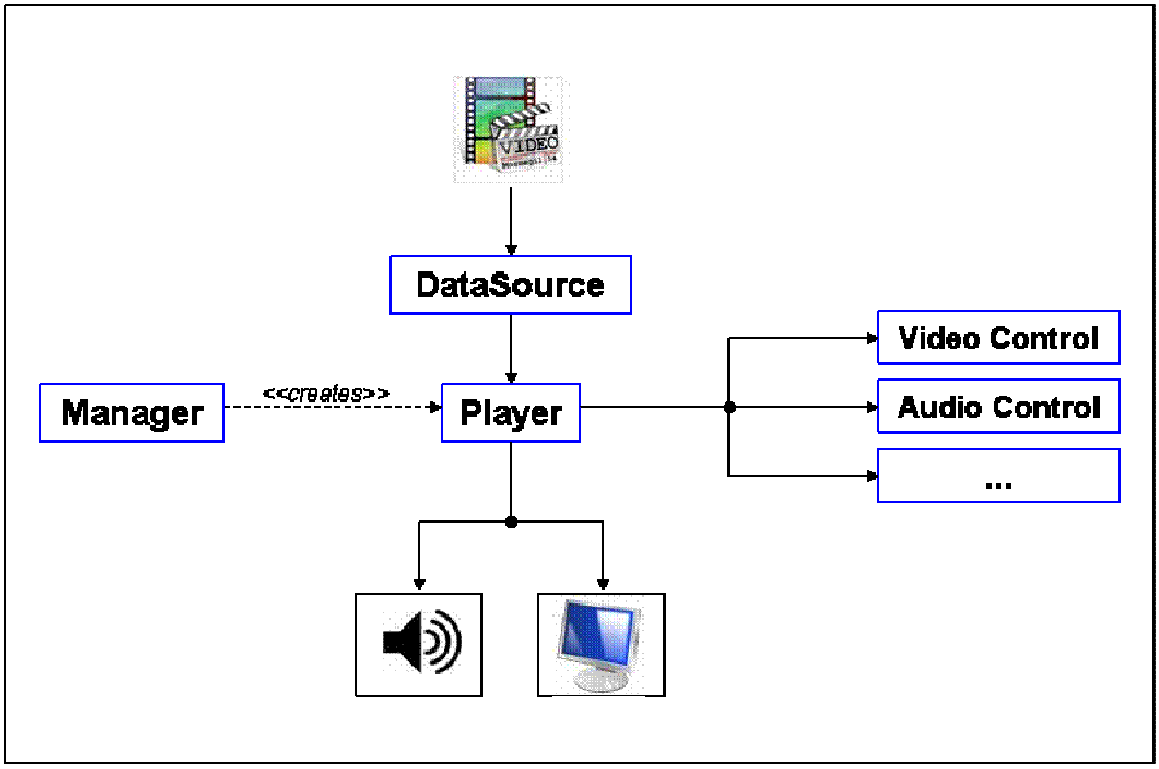


Figure 18. J2ME MMAPI simplified component diagram

6 STREAMING LIBRARY

MIDP 2.0 Media API does not support real-time media streaming, and thus can only be used for media download or media on the local storage. Although the MMAPI specification is designed for real-time streaming and supports custom DataSource, many mobile phones that claim to be MMAPI-compliant do not have real-time streaming capability or extensibility of DataSource.

In this project, a streaming library is developed to provide the streaming capability to these semi-MMAPI-compliant devices. It enables multimedia playback from a remote media source, using an approach different from *Media Download*, mentioned in Section

3.1. It also makes use of the free Bluetooth medium to get rid of the data cost associated with the traditional way of doing networking on mobile devices.

6.1 MOTIVATIONS

There have been many semi-MMAPI-compliant mobile phones on the market. As new devices are manufactured to meet the high demand for multimedia, one would think that most, if not all, of recently made phones would be equipped with real-time streaming. However, reality proves the opposite: many low-end to medium devices are still being produced semi-MMAPI-compliant, due to 1) lack of hardware, 2) manufacturing cost reduction, 3) time-to-market constraint, and 4) technical resources constraint. Thus, it is motivational to bring the multimedia streaming capability to this subset of mobile devices.

6.2 REQUIREMENTS

The library is targeted at devices that do not have streaming support in Java, and thus will need to:

- Provide a module implemented in Java that J2ME applications can use to facilitate multimedia streaming-like capability.
- Provide a Bluetooth proxy implementation to avoid using cellular network.

6.3 ARCHITECTURE DESIGN

The targeted mobile devices already can play local media files encoded in a number of formats, such as MPEG-4, MIDI, etc. The assumption is that they will not need the media decoders, and all the library has to deliver is the protocol handling feature to retrieve the media contents that the device already has the decoders for.

For contents, the media is encoded in MPEG-4 and uses 3GP container format. The streaming library is provided as a module, and the J2ME application sits on top of this module. The following diagram shows the intended use of the library.

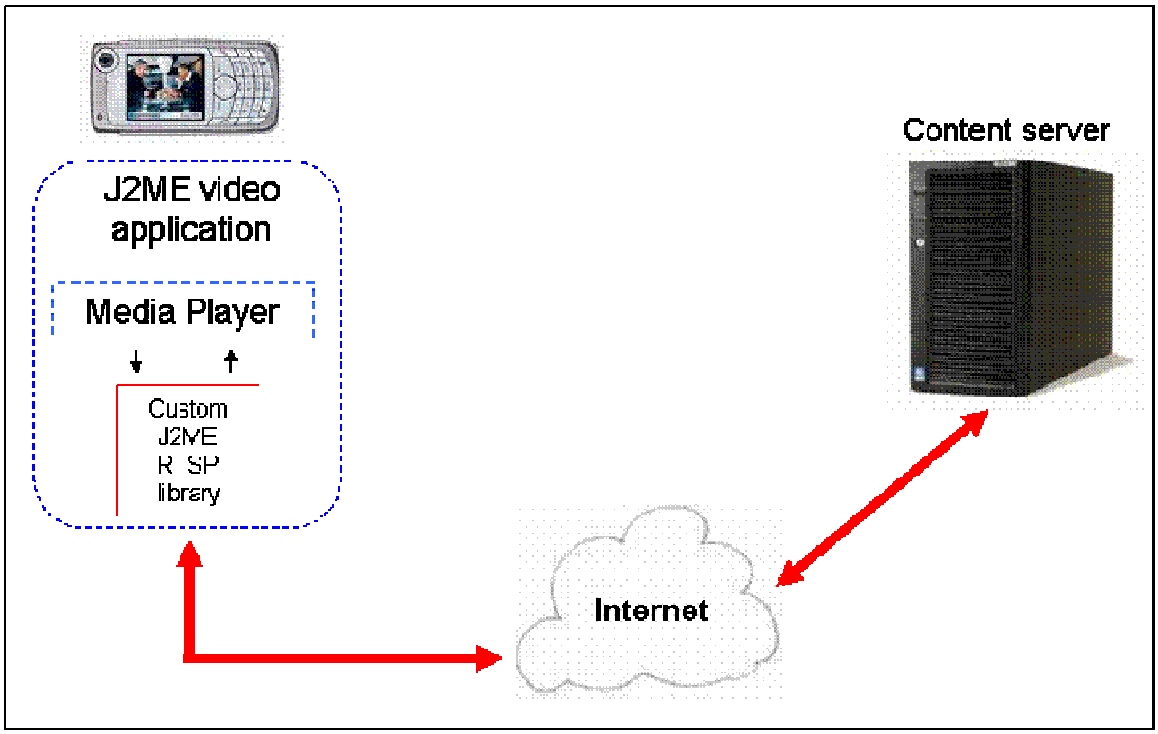


Figure 19. Custom streaming library module

The cellular tower and the carrier gateway are left out for clarity. The module handles all the networking and interaction with the remote server, and supplies the media data to the media player via the J2ME application. Although the diagram shows the library sits below the media player layer, it is actually a part of the J2ME application.

6.3.1 DIRECT DELIVERY USING WIRELESS TCP/IP NETWORK

TCP and UDP are among the protocols supported by GCF. TCP and UPD connections can be established using *socket* and *datagram* in the protocol name as in Section 5.2.1. Using TCP and UDP connection provides a direct mapping from TCP to RTSP channel, and UDP connection(s) to RTP and RTCP, as in the real-time streaming specification.

IP packets are carried over the cellular GPRS data network and forwarded to the streaming server. For new devices with WiFi capability, the phone can join the home wireless network and use the IP network directly. The following diagram shows a mobile device using the WiFi network topology to access the streaming server.

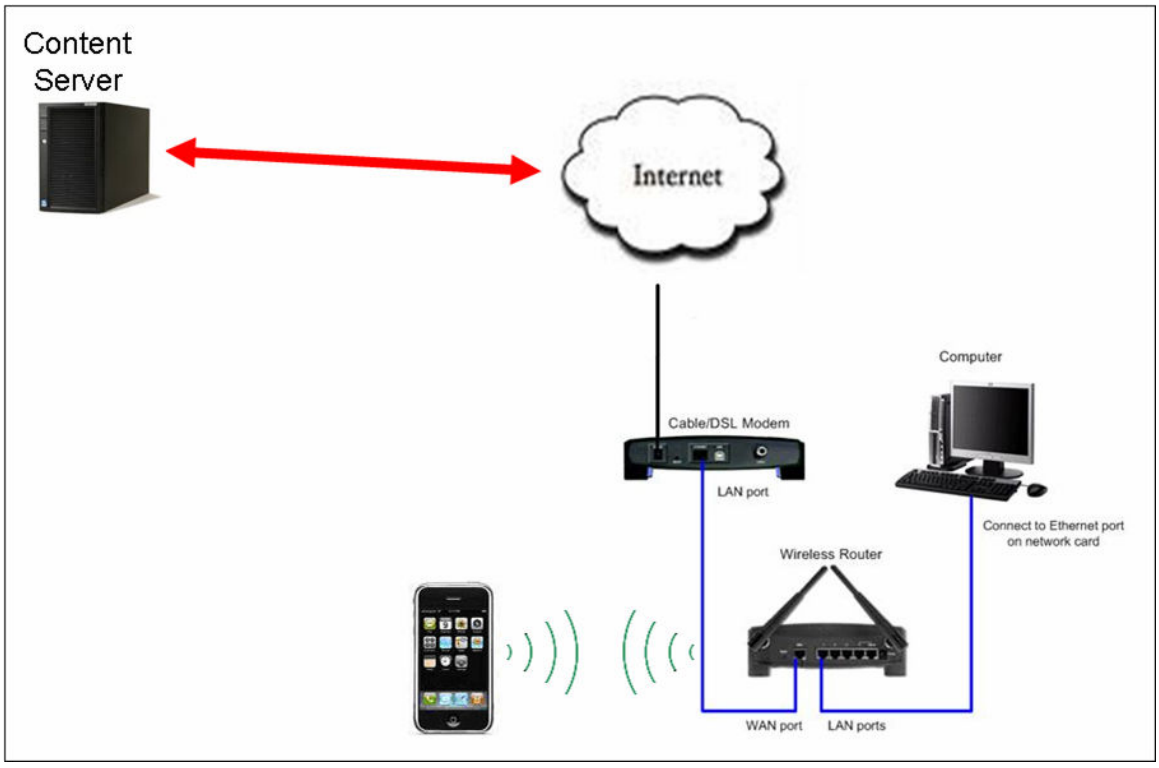


Figure 20. Custom streaming library in a WiFi environment

6.3.2 DELIVERY OVER A BLUETOOTH PROXY

Bluetooth is a short-range and low-speed radio technology used in many mobile gadgets. Bluetooth capability has recently become popular on PC using an add-on USB chip called a dongle. The rationale for using Bluetooth in the streaming library is that a Bluetooth-enabled PC can be used as a proxy, and the local are network (LAN) can be borrowed to deliver media contents from an IP network to a mobile device, for free. Bluetooth is a generic technology which can be used to build other application protocols. To facilitate this usage, a Bluetooth proxy needs to be implemented, which runs on a LAN-connected PC, to interact with the streaming server on the client's behalf. Bluetooth support in J2ME is in JSR 82 specification. The following diagram shows the suggested stacks involved in the streaming-over-Bluetooth approach between a mobile device and a LAN-connected PC.

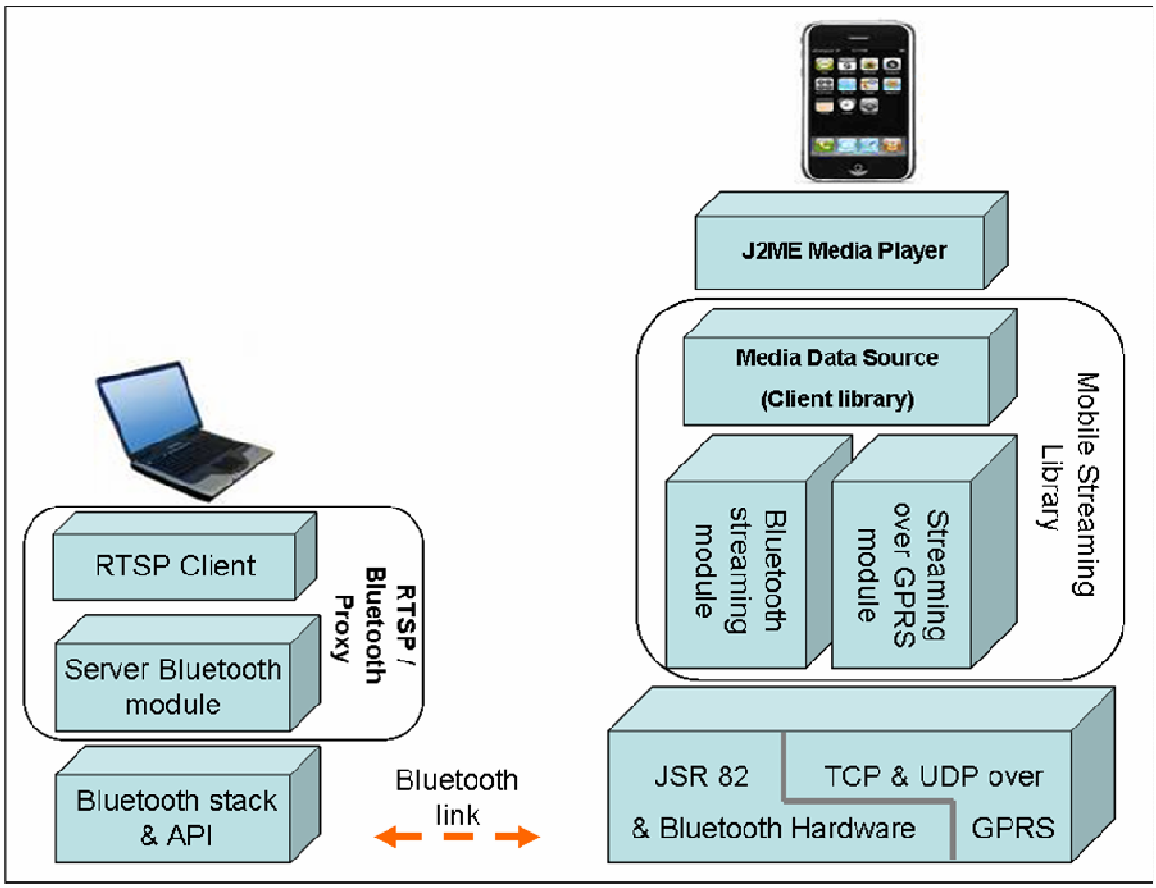


Figure 21. Streaming library and Bluetooth proxy

Even though the LAN side is omitted, it is just a normal LAN connection. The PC can either be connected to a wired LAN, or a wireless (802.11) local network.

6.4 PROJECT DESIGN

Three approaches have been considered and are discussed in details below. The most appropriate method is chosen as the solution. The first two methods use the open-source Darwin streaming server, while the last one uses a custom server.

6.4.1 TCP/UDP APPROACH [10]

This method uses a custom DataSource to implement the streaming protocol. The custom DataSource, *RTPDataSource*, implements the specific delivery protocol, in this case, by using both TCP and UDP. The following diagram shows a simplified view of the classes involved.

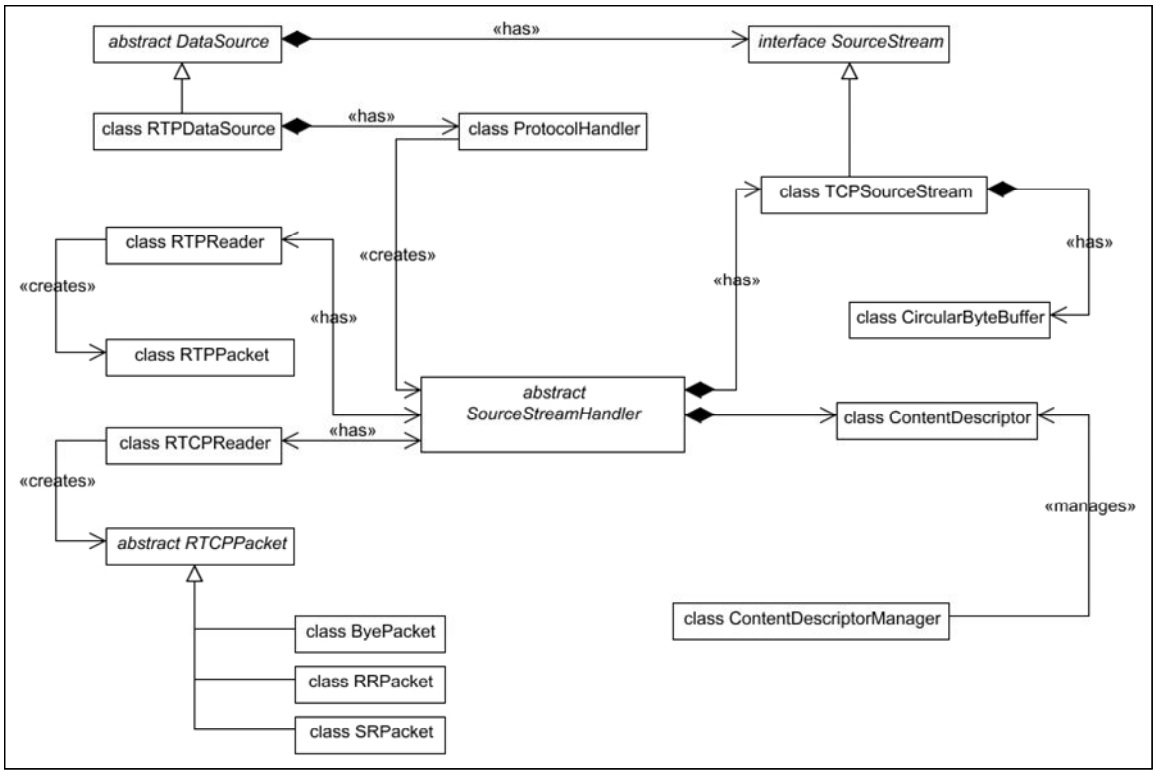


Figure 22. Class diagram for the TCP/UDP approach

In this approach, a TCP connection is used for the RTSP channel, and for each track, a pair of UDP connections is used for RTP and RTCP channels.

The class RTPDataSource implements the interface DataSource, and contains an instance of ProtocolHandler, which handles all the RTSP requests and responses. With knowledge of the media from the DESCRIBE response, ProtocolHandler creates a collection of SourceStreamHandler instances to represent the tracks, with each encapsulating an RTPReader and RTCPReader that receive RTP packet and RTCP packet from RTP and RTCP channels, respectively.

A SourceStreamHandler object contains a TCPSourceStream instance that implements the interface SourceStream. A SourceStreamHandler can be of either audio or video type, and creates the appropriate controls for that track. It also carries track-specific information, such as track id, RTP and RTCP port numbers.

TCPSourceStream uses a CircularByteBuffer object to store the accumulated RTP packet data that will be consumed by the Player. The circular buffer has APIs for reading data from and writing data to it.

An *RTPReader* class is implemented to handle incoming RTP packets. These packets are received, reconstructed, and then dispatched to the owning SourceStreamHandler object. RTPReader is implemented in a Thread to work in a concurrent, non-blocking fashion.

Similarly, an RTCPReader object is used to handle the RTCP channel. It reads and reconstructs RTCP packets, and also forwards them to the associated stream handler.

An RTP packet is represented by an instance of the class *RTPPacket*. This class can decode a binary buffer to reconstruct the RTP packet with properties like payload type, sequence number, timestamp, and the actual data, etc. A handful of other classes are used to represent only the prominent RTCP packet types: Bye, Receiver Report, and Sender Report. Naturally, they are named *ByePacket*, *RRPacket*, and *SRPacket*.

RTPDataSource's operations are directly mapped to the operations on ProtocolHandler, such as *connect(), disconnect()*, *start()*, and *stop()*.

Player	*DataSource*	ProtocolHandler
realize()	connect()	DESCRIBE
prefetch()	xxx	xxx
start()	start()	SETUP, PLAY
stop()	stop()	PAUSE
close()	disconnect()	TEARDOWN

Figure 23. Operation mappings

The table above maps operations from the Player interface to operations on the custom DataSource, and then to operations on ProtocolHandler. ProtocolHandler takes care of the actual networking between client and server, and also keeps track of the session state, such as Described, Setup, Playing, or Stopped. "xxx" means there is no matching operation. These mappings are formed by observing and debugging the media framework.

When *realize()* is called on the Player object , it invokes *connect()* on the custom DataSource. According to the MMAPI specification, *realize()* examines the media data; and according to the media streaming standard, the DESCRIBE request is used to retrieve media information. Thus, these operations are associated together.

"*prefetch()* acquires the scarce and exclusive resources and processes as much data as necessary to reduce the start latency" [11]. However, there is not any way to acquire device's hardware resources, such as speaker and display, and also there is any matching operation on the DataSource interface, it is not associated with any operation (no-op).

The operation *start()* on Player calls *start()* on DataSource to resume the data transfer. At this point, we need to send the SETUP request if we have not done so, then send the PLAY request to begin or resume the data transfer.

The operation *stop()* on Player calls *stop()* on DataSource, which "will pause the playback at the current media time" [11]. Therefore, ProtocolHandler sends PAUSE to the server. And finally, *close()* – used to close the Player and release all resources – is mapped to *disconnect()* on DataSource, which causes the TEARDOWN request to be sent by ProtocolHandler.

This approach adheres well to the real-time multimedia streaming specification. However, it does not work at the last mile of the cellular network. The problem this implementation faces is the cellular carrier, being very conscious of the security risks involved in opening UDP ports, has blocked all incoming UDP traffic at their IP gateway. UDP packets destined for the phone are not forwarded to the device. The phone can send RTSP commands to the Darwin server, and receive RTSP responses successfully. By sniffing the traffic on the server with Ethereal, we can see that RTP and RTCP packets are sent out by Darwin server, but none of them reaches the mobile device. This is confirmed by testing in the emulator running on a PC. The UDP connections can be established between the client running on the emulator and Darwin server. The client library can also receive UDP packets originating from the server machine. Appendix A shows the detailed class diagram of the TCP/UDP approach.

6.4.2 TCP-INTERLEAVING APPROACH [5]

Since UDP packets are not forwarded by the carrier's gateway, we switch to using TCP-interleaving. As discussed in Section 4.2.2, TCP-interleaving uses one TCP connection to multiplex RTSP request/response and RTP/RTCP packets. The challenge then lies in the de-multiplexing of data and reconstructing the packets for the various channels.

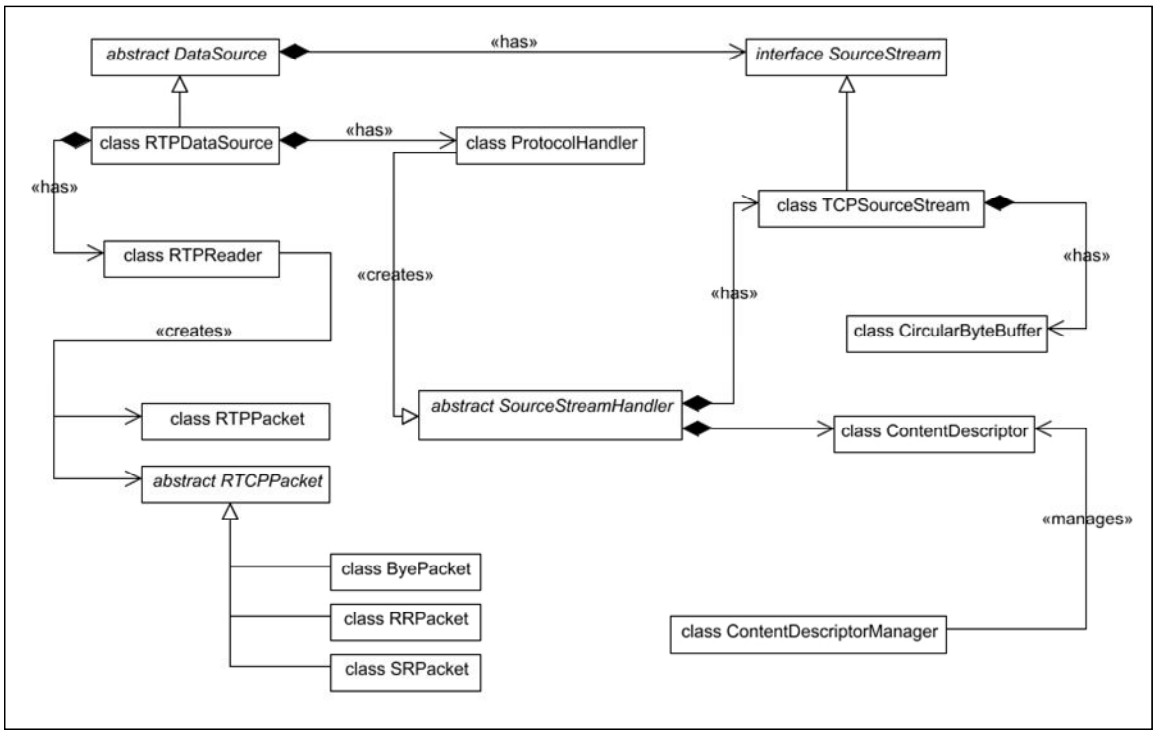


Figure 24. Class diagram for the TCP-Interleaving approach

The class diagram for this approach resembles that of the TCP/UDP. We have the familiar RTPDataSource entity implementing the *DataSource* interface, and also the TCPSourceStream class implementing the *SourceStream* interface. Again, TCPSourceSgtream contains a CircularByteBuffer object, used to buffer media data.

Similarly, we have the ProtocolHandler class bridging the custom DataSource and TCPSourceStream, through the abstract class SourceStreamHandler. Each SourceStreamHandler instance manages a track in the media.

There is no longer an RTCPReader entity. The class RTPReader now reads and recreates both RTP and RTCP packets, since they are sent on the same connection. The

TCP connection is also shared with ProtocolHandler, which needs to send and receive RTSP request and responses.

```
SETUP rtsp://mstream.dyndns.org:554/crazydancing.3gp/trackID=65536 RTSP/1.0
CSeq: 2
TRANSPORT: RTP/AVP/TCP;interleaved=0-1


RTSP/1.0 200 OK
Server: DSS/5.5.4 (Build/489.13; Platform/Win32; Release/Darwin; )
Cseq: 2
Last-Modified: Fri, 12 Jan 2007 06:32:32 GMT
Cache-Control: must-revalidate
Session: 91061896640550
Date: Wed, 28 Feb 2007 04:11:40 GMT
Expires: Wed, 28 Feb 2007 04:11:40 GMT
Transport: RTP/AVP/TCP;interleaved=0-1
```

Figure 25. Sample interleaved SETUP request/response

The example above shows the SETUP request and response to establish an interleaving session. The special parameter "*interleaved=0-1*" is used to request that RTP and RTCP data be interleaved on the same connection, identified by channel number: 0 for RTP and 1 for RTCP. This same parameter is sent back in the response to acknowledge the request.

The same convention of using even channel numbers for RTP and the next higher odd numbers for RTCP is followed. A track thus owns two channel numbers, an even for RTP channel and an odd for RTCP channel. RTPReader does not know anything about the different tracks. It can only tell if the interleaved data is an RTP packet or an RTCP packet, by looking at the channel number parity: even channel number – RTP packet; odd channel number – RTCP packet. It parses the binary data, reconstructs the packet, and sends it to RTPDataSource, along with the channel number. RTPDataSource then forwards this packet and the associated channel number to ProtocolHandler for packet handling.

ProtocolHandler needs to keep track of which SourceStreamHandler owning which channels. It maintains this information in two hash tables, mapping from channel number

to SourceStreamHandler object. Using the reported channel, ProtocolHandler looks up the correct SourceStreamHandler object and passes the packet to it.

SourceStreamHandler receives the packet, and if it is an RTP packet, sends it to TCPSourceStream for buffering. The media data encapsulated in the RTP packet is removed and inserted into the circular data buffer.

The operation mappings between Player, DataSource, and ProtocolHandler are the same as in the TCP/UDP approach. Appendix B shows the complete TCP-interleaving class diagram.

Since this approach does not use UDP connection, it succeeds in getting the RTP and RTCP packets delivered to our streaming library. Like before, we run Darwin server on the server PC, and use Ethereal to sniff the network packets. Three tests are conducted for comparison, using:

1. ***Native streaming on Sony Ericsson W850i:*** Since we cannot tap into the native streaming code, we have to sniff for network traffic on the server machine. The UDP packets (column #1) are captured on the server using Ethereal, and recorded in the first test case. We inspect each UDP packet for the length of the encapsulated RTP packet (column #2), and the length of the media data contained in that RTP packet payload (column #3).
2. ***TCP-Interleaved on emulator:*** Using the client streaming library in the J2ME application on the emulator, we can print out each RTP packet that RTPReader constructs from the interleaved data. Since UDP is not used, we record only the RTP packet length (column #4) and media data length (column #5) in the payload.
3. ***TCP-Interleaved on W850i:*** This time, the same test as in (2) is conducted, but on the real device. Running the same application on the Sony Ericsson W850i, we log each incoming RTP packets in the phone's local storage, and upload the log file to the server at the end of the session for viewing. Again, the RTP packet length and payload length are recorded in the log, shown in columns #6 and #7, respectively.

The total number of RTP packets sent and received matches in three tests, 96 packets. For RTP packets, the octets sum up to be 68,647 bytes in all three cases; and so do the sums of the payload octets, at 67,496 bytes. Although we have not inspected and compared each packet's content, we are confident that the client streaming library using TCP-interleaving is working properly. Appendix C shows the complete recordings of the three tests described above.

Media data encapsulated in RTP packets can now be collected and properly inserted into the source streams. However, we get into a different issue: the phone is only semi-MMAPI-compliant. The *Player* object, although being given a custom *DataSource* object, just treats it like it does with a local input stream. When local input stream is used, the entire media file is loaded into memory and the memory buffer is passed to the player. A custom DataSource being treated like an input stream is no difference from whole media download. A true MMAPI-compliant implementation fetches data from the DataSource, little by little, and presents it when just enough data has been buffered. Three high-end phones, a Nokia and two Sony Ericsson devices, have been tested with TCP-interleaving. Not only that none of the tested phones provides true MMAPI implementation with custom DataSource support, but also none of the mobile phones on the market is reported to be fully MMAPI-compliant.

6.4.3 MULTI- SUBCLIP APPROACH

In this approach, the intended media file is broken into smaller sub-clips - each is complete by itself and can be played independently. The original media file can be split based on duration or sub-clip file size using *Mp4box* from the GPAC tool suite. The shorter the sub-clips are (either in duration or size), the less time it takes to download the individual clips. However, this method is similar to playing a multi-disc movie: we have to remove the current disc and insert in the next one. This process takes time, and there is a gap in between the sub-clips. Thus, short clips reduce download delay at the cost of increasing gaps between clip playbacks.

Mp4box from the GPAC tool suite is used to split the following media file into ten-second clips. There really is no rule on how long each clip should be, either in time duration or size. Experiments are conducted with the splitting process to find the most

optimal configuration - one which does not produce: 1) a long download delay per clip and 2) the slideshow-effect because of fast switching among very short clips. To satisfy these two criteria, the clips should be as long as possible (in terms of file size) without sacrificing clip download time. Having shorter clips also means there are more clips per video, which in effect creates a bigger description file.

In this experiment, the demo video is split into 10-second subclips of varying sizes. The choice of using a fixed clip-duration rather than fixed clip-size is made to ensure that the flashing effect is only seen at regular intervals. This choice helps demonstrate the clip switching effect, and may also enhance user experience. Also, since videos targeting mobile devices are most likely encoded using fixed bit-rate, splitting based on duration produces little variation in clip size, as in the following example, where clip sizes range from 72.3 KB to 79.2 KB.

```
mp4box -split 10 KRON.3gp

Storing split-file KRON_001.3gp - duration 10.00 seconds
[3GPP convert] Setting major brand to 3GPP V5 file
Storing split-file KRON_002.3gp - duration 10.00 seconds
[3GPP convert] Setting major brand to 3GPP V5 file
Storing split-file KRON_003.3gp - duration 10.00 seconds
[3GPP convert] Setting major brand to 3GPP V5 file
Storing split-file KRON_004.3gp - duration 10.00 seconds
[3GPP convert] Setting major brand to 3GPP V5 file
Storing split-file KRON_005.3gp - duration 10.00 seconds
[3GPP convert] Setting major brand to 3GPP V5 file
Storing split-file KRON_006.3gp - duration 8.96 seconds
[3GPP convert] Setting major brand to 3GPP V5 file
```

Figure 26. Splitting video file using mp4box (GPAC)

The client library will need some knowledge about the media and the associated clips. We design a custom media description language, mirroring SDP. Darwin streaming server can no longer be used. Instead, a server is developed to handle request for a media file and deliver the sub-clips in response. A custom RTSP language is

created to enable the communication between the client and the custom server. Our custom RTP is used to deliver media data. It does not really have a format, and will be addressed as the RTP channel instead.

6.4.3.1 NAMING CONVENTION

To distinguish between our custom formats and the various standards defined in the RFCs, we will append the letter *"c"* to the custom format, i.e., SDP-c, and append the RFC number to the standard format, i.e., SDP-4566.

6.4.3.2 CUSTOM SDP

RTSP uses the SDP standard defined in RFC 4566 to convey the media information, such as type and format, media transport and session description metadata to the client. The following snapshot shows the SDP-4566 media description for a video file named CrazyDancing.3gp.

```
v=0
o=StreamingServer 3381624698 1168583552000 IN IP4 172.16.1.39
s=\crazydancing.3gp
u=http:///
e=admin@
c=IN IP4 0.0.0.0
b=AS:80
t=0 0
a=control:*
a=x-copyright: MP4/3GP File hinted with GPAC 0.4.3-DEV (C)2000-2005 -
http://gpac.sourceforge.net
a=range:npt=0-   5.67300
m=video 0 RTP/AVP 96
b=AS:67
a=rtpmap:96 H263-1998/90000
a=control:trackID=65536
a=cliprect:0,0,144,176
a=framesize:96 176-144
m=audio 0 RTP/AVP 97
b=AS:13
a=rtpmap:97 AMR/8000/1
a=control:trackID=65537
a=fmtp:97 octet-align
```

Figure 27. Sample SDP-4566 packet

The SDP-4566 syntax above tells the session name in *"s=\crazydancing.3gp"*, connection information in *"c=IN IP4 0.0.0.0"*, the two media streams: one video in *"m=video 0 RTP/AVP 96"* and one audio in *"m=audio 0 RTP/AVP 97"* along with their track ids. The audio and video lines also describe the encoding, such as AMR sampled at 8 KHz, and H263 video format of size 176x144.

Our streaming client is interested only in the media information, but not the protocol and session description. It has to know the media format, and since the original media file is split into smaller clips, it also needs to know the number of clips constituting the requested media, and for each clip, its duration and size. The clips must be requested and played in the correct order so that they form a continuous media session. These requirements result in the following custom SDP-c format:

```
m=<media-name>
t=<media-type>
p=1 <media-part-001> <start-time-ms> <duration-ms> <length-bytes>
p=2 <media-part-002> <start-time-ms> <duration-ms> <length-bytes>
p=n <media-part-00n> <start-time-ms> <duration-ms> <length-bytes>
```

Figure 28. Custom SDP format

The attribute *"m="* gives the name of the original media that this SDP describes. The attribute *"t="* tells the media format. The *"p="* lines list all the sub-parts of the media. Each *p*-line contains the (clip) part id, followed by the part name, the starting time in milliseconds, the part duration in milliseconds, and finally the length of that part in bytes. The figure below shows the media description for a video named KnightRider.3gp in the SDP-c format.

```
m=KRON.3gp
t=video/3gpp
p=1 KRON_001.3gp 0 10000 76284
p=2 KRON_002.3gp 10000 10000 79240
p=3 KRON_003.3gp 20000 10000 75326
p=4 KRON_004.3gp 30000 10000 75030
p=5 KRON_005.3gp 40000 10000 77828
p=6 KRON_006.3gp 50000 9750 72352
```

Figure 29. Sample custom SDP packet

The video KnightRider.3gp has a type of "video/3gpp" and consists of six parts, starting with KnightRider_001.3gp as the first clip, followed by KnightRider_002.3gp, etc. and ending with KnightRider_006.3gp. For each part, the duration in milliseconds and length in bytes are also included, i.e., 9,472 ms with a length of 72,589 bytes for the first sub-video, which starts at time 0.

The SDP–c-formatted information is encapsulated in the DESCRIBE response, to tell the client what it needs to request, and in which order. This custom SDP format offers high flexibility like the original SDP specification, allowing more attributes to be added. If extensibility is required, the client library, and possibly the server, will need to be changed to understand the new information that it may be interested in.

6.4.3.3 CUSTOM RTSP

A new set of RTSP syntaxes are also developed to fit the customized protocol. Mirroring the conventional RTSP specification, the new protocol has the following requests:

- DESCRIBE: to obtain the media meta-data in the custom SDP format.
- SETUP: to obtain a unique session id that is used to associate a client with a session and network connections.
- PLAY: to request a specific sub-clip.
- TEARDOWN: to terminate the session with the server.

As each clip is completely playable by itself and resides in the client's memory, there is no need for a PAUSE command. The client can just issue a local Pause request when needed.

All RTSP-c commands follow the RTSP-2326 command syntax by including a *"Cseq:"* header, used to pair the request and reply. Lines in request and response also end with a pair of CarriageReturn/LineFeed (CRLF), and the request and response is terminated with two pairs of CRLF.

The DESCRIBE request contains the command *"DESCRIBE"* followed by the media request URL and the RTSP version. Media request URL is in the familiar format *rtsp://host.domain:[port]/media-file*, where *port is optional*.

The client issues the SETUP request to obtain a session id used in successive requests. The session id is used by the server to associate the different channels, in case of using TCP/IP, and to locate and identify the different clients. The SETUP request also starts with REQUEST as the command, followed by the rtsp URL and ended with rtsp-version.

The PLAY request and reply require more headers to convey information about the requested media clip. Below is the format for the PLAY request:

```
PLAY <media-URI> RTSP/1.0
CSeq: <sequence-number>
Session: <session-id>
File: <media-part-name>
FileSize: <media-part-size>
```

Figure 30. RTSP-c PLAY request

The TERMINATE request serves the same purpose as in RTSP-2326. Upon receiving this command, the server replies with an OK, terminates all connection channels with client, and cleans up resources used to service that client session.

Figure 31 shows a complete streaming session, using the custom SDP and RTSP, which plays only the first clip of the media. It shows where SDP-c is used along with all the commands described in this section.

6.4.3.4 CUSTOM RTP CHANNEL

The multimedia streaming standard uses RTP packets to carry media data, possibly in a separate channel. Our custom streaming design does not break media data into packets,

but rather delivers unit of whole clips. RTP-c channel specifies only the API, and the implementation is dependent on the underlying network protocol in use.

For network supporting multiple connections, such as IP, a separate connection can be established for the RTP channel. However, for network topology that allows only a single connection between two devices, such as Bluetooth, the same Bluetooth connection is used to carry both RTSP and RTP traffics. In this case, RTP is implemented as a virtual channel, rather than a dedicated and physical channel.

The RTP-c API is designed to be very simple. The calling library uses it to send a session id so the remote server can pair it with the corresponding RTSP channel, if separate connections are used. The caller also uses the RTP-c channel to read back a stream of media data after a successful PLAY response.

6.4.3.5 CLIENT MEDIA MANAGER

At the high-level, there are three tasks associated with the multi-clip approach:

- Maintaining media information: such as the various clips, clip order, clip start-time, duration and length. This is required to maintain a continuous stream of clips to form the original media.
- RTSP protocol handling: to send and process RTSP request and response, as well as to retrieve clip data.
- Managing clip playback: each clip is played independently, but in a series to form a longer piece of multimedia.

All the above tasks are dealt with using a media manager. The media manager is the entry point to this method of streaming, from the application's point of view. Applications do not know anything about SDP-c, RTSP-c, and RTP-c. All they know about is the media manager and the media manager does the all the work associated with the tasks described above.

```
DESCRIBE rtsp://mstream.dyndns.org/KRON.3gp RTSP/1.0
CSeq: 1


RTSP/1.0 200 OK
CSeq: 1
Content-length: 237

m=KRON.3gp
t=video/3gpp
p=1 KRON_001.3gp 0 10000 76284
p=2 KRON_002.3gp 10000 10000 79240
p=3 KRON_003.3gp 20000 10000 75326
p=4 KRON_004.3gp 30000 10000 75030
p=5 KRON_005.3gp 40000 10000 77828
p=6 KRON_006.3gp 50000 9750 72352


SETUP rtsp://mstream.dyndns.org/KRON.3gp RTSP/1.0
CSeq: 2


RTSP/1.0 200 OK
CSeq: 2
Session: a762310f-9ecc-4d94-9c6f-ecf6ef22be5c


PLAY rtsp://mstream.dyndns.org/KRON.3gp RTSP/1.0
CSeq: 3
File: KRON_001.3gp
FileSize: 76284
Session: a762310f-9ecc-4d94-9c6f-ecf6ef22be5c


RTSP/1.0 200 OK
CSeq: 3
Session: a762310f-9ecc-4d94-9c6f-ecf6ef22be5c


TEARDOWN rtsp://mstream.dyndns.org/KRON.3gp RTSP/1.0
CSeq: 6
Session: a762310f-9ecc-4d94-9c6f-ecf6ef22be5c


RTSP/1.0 200 OK
CSeq: 6
```

Figure 31. Custom RTSP streaming session

6.5 PROJECT IMPLEMENTATION

6.5.1 SHARED MODULE(S)

The streaming client library and streaming serve are implemented in Java. The server uses Java 2 Standard Edition (J2SE), while the client library uses J2ME. Since J2ME is a subset of J2SE, some components shared by both the client and the server are implemented as shared modules. Three shared modules are identified: SDP, RTSP, and a utilities module.

6.5.1.1 SDP MODULE

This module consists of only one class: DML, standing for Descriptor Markup Language. It contains the structure for the SDP-c format, with properties to keep track of media name and type (*"m="* and *"t="* lines) and entries to keep track of individual clips (*"p="* lines). It can parse and interpret the textual form of SDP-c contained in the DESCRIBE response, as well as build a textual representation of the SDP-c format to send in the same response.

6.5.1.2 RTSP MODULE

This module consists of the following classes:

- RTSPException: a generic exception class used in RTSP request/response parsing.
- RTSPTypes: serves as the base class for RTSPRequest and RTSPResponse. It declares RTSP header definitions, such as *"DESCRIBE"*, *"SETUP"*, *"PLAY"*, *"CSeq:"*, *"Session:"*, *"File:"*, *"FileSize:"*, and RTSP version, etc. It also has API to add and retrieve the header values.
- RTSPStatusCode: maintains a list of error codes and error message mapping that are consistent with the RTSP-2326 standards.
- RTSPRequest: derives from RTSPTypes and represents an RTSP request. It encapsulates the commands specific to a request, such as "DESCRIBE".
- RTSPResponse: also derives from RTSPTypes and encapsulates an RTSP response.
- RTSPReader: an interface that defines APIs for reading lines of string from an RTSP channel. It is used by RTSPRequest and RTSPResponse. Network

protocol-specific implementation, such as TCP or Bluetooth, will derive from this interface.

- RTSPWriter: defines APIs for writing a string to an RTSP channel, and is used by RTSPRequest and RTSPResponse. Similar to RTSPReader, network protocol-specific derived classes, such as TCP and Bluetooth, will provide the implementation.
- RTSPSocketReader: socket-implementation for RTSPReader, used by both server and client socket library.
- RTSPSocketWriter: socket-implementation for RTSPWriter, used by both server and client socket library.

6.5.1.3 UTILITIES MODULE

This module defines classes used for debugging, logging, and string manipulations.

6.5.2 STREAMING SERVER

The server, implemented in the class StreamingServer, uses TCP/IP connections for communication and media delivery. The server contains two threaded services, to concurrently support multiple client sessions. The following figure shows the class diagram of the streaming server.

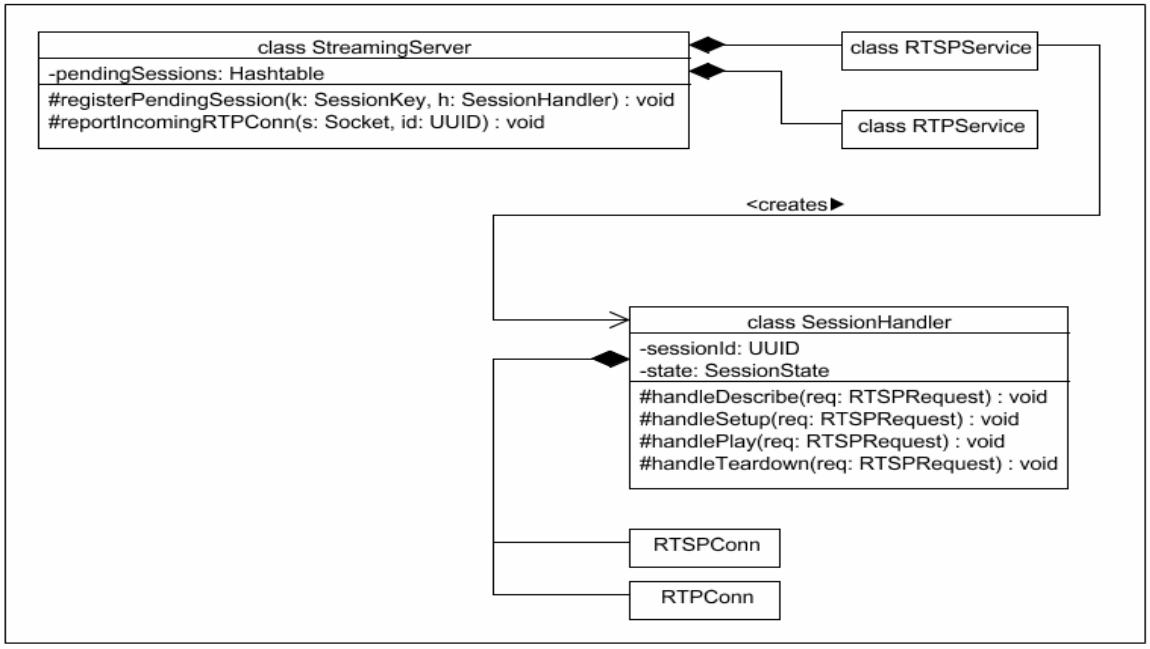


Figure 32. Custom streaming server class diagram

The RTSP service, implemented in the class RTSPService, binds to TCP/IP port 5454 and listens for new client requests. When a new session request arrives, RTSPService creates a new SessionHandler object to handle all of client's RTSP requests. SessionHandler is implemented as a thread so it can run on its own. When the client sends a DESCRIBE request, SessionHandler checks its media collection to see if the requested media exists. If the requested media is present, the handler uses a DMLParser object to read the media's DML file. DMLParser understands the format and constructs the DML object, then returns it to SessionHandler. SessionHandler replies with an RTSP OK, along with the SDP-c formatted DML. When SETUP is received, SessionHandler generates a unique session id and sends it in the response. At the same time, it registers itself with the streaming server as a pending session, using a session key. The session will become complete when the RTP channel connection is established successfully.

The class RTPService provides the implementation for the RTP service, which listens on TCP port 5455. As a TCP connection is opened, it accepts and adds it to the read connection list of a Selector object (Java implementation of the Unix *select()* method). The client is expected to send the session id next, and the service reports the new connection and the session id to the streaming server. The streaming server then compares the session id with those in the pending session list, and if a match is found, the session is fully established and is removed from the list.

The two established TCP connections are now owned by SessionHandler, which is responsible for the streaming session from now on. RTSP requests and responses are sent over the RTSP connection, and media data is sent over the second connection.

Figure 33 shows the main body pseudo-code of SessionHandler.

```
while (sessionState != SessionState.CLOSED)
{
   req = rtspConn.getRequest();
   if (req == null)
      acknowledgeBadRequest();
   else
   {
      switch (req.getRequestType())
      {
         case RTSPRequest.DESCRIBE_REQ:
            handleDescribe(req);
            break;
         case RTSPRequest.SETUP _REQ:
            handleSetup(req);
            break;
         case RTSPRequest.PLAY _REQ:
            handlePlay(req);
            break;
         case RTSPRequest.TEARDOWN _REQ:
            handleTeardown(req);
            break;
      }
   }
}
```

Figure 33. SessionHandler body

There are other classes in the server module providing the implementations for RTSP and RTP connections. These classes are RTSPSocketConn and RTPSocketConn, respectively, which provide the APIs for receiving an RTSP request and sending the response, and to stream the requested media's contents. The module also has logging support used for monitoring and debugging purposes.

Figure 34 shows the server receiving and responding to a DESCRIBE and a SETUP requests. This session is established using the Bluetooth client implementation. The client of this session is the streaming proxy, which will relay this information back to the actual client - the mobile device.

Figure 34. Streaming server in operation

6.5.3 CLIENT STREAMING LIBRARY FRAMEWORK

The client library is designed for extensibility with different network technologies. The framework is broken down into packages and interfaces so that a new implementation specific to a network protocol can be easily added.

The client streaming framework incorporates and extends the shared library module discussed above. It consists of six main packages, outlined below:

1. ***com.mstream.client***: contains UI component implementations for the sample streaming application. These include the main application, the two video screen types (VideoCanvas and VideoForm), and SessionChooser and SessionChooserListener. The video screen types are used for video display. The SessionChooser class is for selection one of two supported session

implementations: Socket or Bluetooth. The following classes belong to this package:

- VideoScreen: base class for VideoCanvas and VideoForm.
- VideoCanvas: canvas-based video display implementation.
- VideoForm: form-based video display implementation.
- SessionChooserListener: provides a listener interface to notify when a session is selected for use.
- SessionChooser: provides the UI component for session browsing and selection.
- StreamingClient: the sample video streaming application.

2. ***com.mstream.client.io***: this package contains the connection interfaces used for RTSP and RTP channels, a base class for the streaming session (StreamingSession), and the session factory. The session factory is used to create Bluetooth streaming sessions. The classes in this package are:
   - RTPConnection: provides an interface for an RTP channel with APIs to send a session id and retrieve the requested media stream.
   - RTSPConnection: abstracts the different RTSP channel implementations.
   - StreamingSession: a base class representing a streaming session, which contains an RTSP- and an RTP connection objects.
   - SessionFactory: produces Bluetooth streaming session objects.
3. ***com.mstream.client.io.bluetooth***: is the Bluetooth implementation of the streaming client. It contains Bluetooth-specific classes that extend the classes and/or implement the interfaces in the two packages above. It contains the following classes:
   - BluetoothServiceInfo: an interface defining L2CAP message types and the UUID of the Bluetooth proxy service.
   - ProxyServiceDiscovery: a utilities class used to discover the Bluetooth proxy services in proximity.
   - ServiceDiscoveryListener: utilities classes used for proxy service discovery.
   - L2CapSession: provides the Bluetooth-implementation for the StreamingSession class.

- RTSP_RTP_L2CapConn: the implementation for both the RTP and RTSP connection interfaces. It handles RTSP request/response and media retrieval.

4. ***com.mstream.client.io.socket***: provides the streaming client implementation using socket, consisting of the following classes:
   - RTSPSocketConnection: implements the RTSP connection using socket.
   - RTPSocketConnection: implements the RTP connection interface using socket.
   - SocketSession: provides the StreamingSession implementation using RTSPSocketConnection and RTPSocketConnection.
5. ***com.mstream.client.util***: contains utilities classes.
   - BufferOverflowException: an exception class used by the CircularByteBuffer and CircularCharBuffer.
   - CircularByteBuffer: implements a circular buffer of bytes.
   - CircularCharBuffer: implements a circular buffer of characters.
   - UUID: provides unique identifiers.
   - Task: an interface representing a unit of work.
   - TaskDispatcher: implements a thread to execute and perform the tasks.
6. ***com.mstream.client.media***: the core framework for the multimedia streaming library. It defines the classes related to media handling and playback, and consists of:
   - DeviceConfig: keeps device-specific information, such as how many open players the device can support.
   - SessionManager: produces sequence numbers (CSeq) for RTSP requests.
   - MediaInfo: maintains information about various tracks in a media.
   - MediaPartPlayer: handles the playback of a single sub-clip.
   - MediaPlayerListener: listener interface for various media playback event.
   - MediaPlayer: combines all the classes and interfaces above to support media streaming and playback.
7. ***com.mstream.test***: provides logging framework and other tests used during development.

- Logger: the logging utilities class.
- RMSLogger: a J2ME application for viewing logs and sending them to a remote server.
- NetworkTest and ProxyServiceDiscoveryTest: network test application and Bluetooth proxy service discovery test.

The following is the class diagram linking the important classes together.

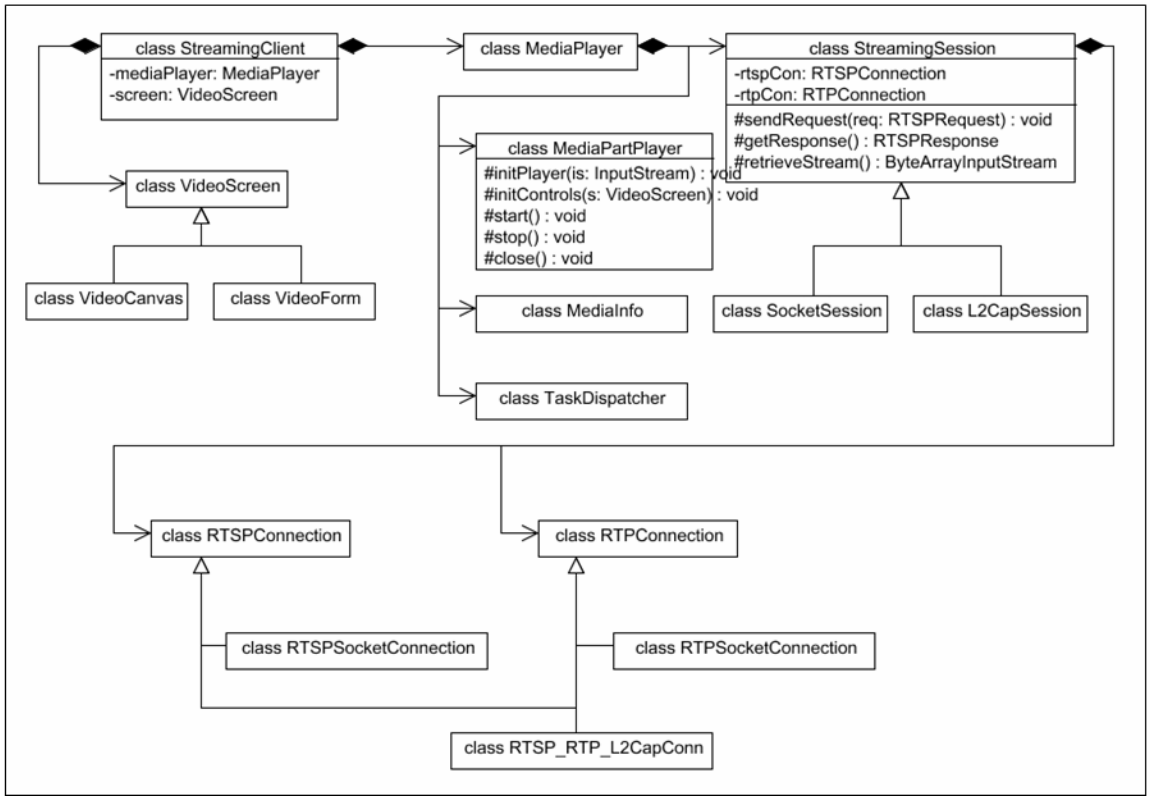


Figure 35. Client library class diagram

Notice the class RTSP_RTP_L2CapConn, which implements both the RTSPConnection and RTPConnection interface. L2CAP permits one active connection between two Bluetooth devices; and so this implementation must multiplex both channels on one physical connection to handle both RTSP and RTP traffic.

6.5.3.1 TCP/IP CLIENT LIBRARY

TCP/IP client library is provided in the package *com.mstream.client.socket*. We use TCP sockets to implement the two channels, RTSP and RTP, separately. The RTSP

implementation is in the class RTSPSocketConnection, and the implementation for RTP channel is in the class RTPSocketConnection.

The RTSP channel opens a socket connection and connects to port 5454 on the remote streaming server. RTSPSocketConnection makes use of RTSPSocketReader and RTSPSocketWriter, provided in the shared module, to send an RTSP request and read back an RTSP response.

The RTP channel connects its TCP sockets to port 5455 on the server. The session id is sent over this socket to the server, and media is read back from this connection.

SocketSession provides the implementation for StreamingSession. It encapsulates both RTSPSocketConnection and RTPSocketConnection, and forms a uniform interface to service the higher layer, the MediaPlayer, to retrieve media data.

6.5.3.2 BLUETOOTH LIBRARY USING L2CAP

Logical Link Control and Adaptation Protocol (L2CAP) is used for the Bluetooth implementation. L2CAP is a packet-based protocol [12, 14, 15], and with a header length of 2 bytes, each packet can be 64 KB in length. However, mobile devices in practice support much smaller packet size of 672 bytes.

Since only one channel can be active between two devices, both the RTSP and RTP channels have to share an L2CAP connection. This means the data packets have to be marked to be either an RTSP or RTP packet. The following message types are defined to identify packets and the data in the payload:

- RTSP_REQ_MSG: an L2CAP packet containing an, or part of an, RTSP request.
- RTSP_RESP_MSG: an L2CAP packet containing an, or part of an, RTP response.
- RTP_MSG_MARKER: an RTP packet containing media data.
- SESSION_ID_MARKER: a packet containing the session id from client to server when establishing the RTP channel.
- CONTINUATION: containing data following the first packet if the data exceeds one packet.

The L2CAP messages are formatted as below:

1-byte	4-bytes	
TYPE	DATA-LENGTH	DATA

Figure 36. L2CAP message format

This message format allows an L2CAP packet to carry a maximum of 667 bytes of data.

Service discovery is abstracted in the class ProxyServiceDiscovery. When requested, this class does an asynchronous discovery and listens for new devices and services. ProxyServiceDiscovery maintains a cached list of found services. When a service is found, it is added to this service cache, and the list can be returned upon request without blocking.

At start-s up, the client instantiates a SessionFactory object. The SessionFactory object then makes a request to ProxyServiceDiscovery to discover the Bluetooth devices in its surroundings that offer the streaming service, using the service id "*5c3d0cd51ec84b0197120b9e1f813d40*". Each service comes with a Bluetooth URL in the following format:

btl2cap://<BDA>:PSM;[param=value;] where

BDA: Bluetooth device address of the device offering the service.

PSM: Protocol Service Multiplexor – used to determine the higher level application protocol.

For each of the found services, SessionFactory creates an L2CapSession object using the associated URL. L2CapSession provides the StreamingSession implementation using Bluetooth technology and represents the remote Bluetooth service.

The core of this implementation resides in the class RTSP_RTP_L2CapConn, which implements both RTSP and RTP operations. This class encapsulates an L2CapConnection instance (defined in JSR 82) and uses two internal classes: RTSPBluetoothReader and RTSPBluetoothWriter, to read and reconstruct an

RTSPResponse object, and to write an RTSPRequest string. All of these three classes act on a single connection, and thus need to control access to the connection using a shared lock.

RTSP_RTP_L2CapConn uses a CircularByteBuffer and a CircularCharBuffer to save data that are read in by mistake. The circular character buffer is used to save read-ahead RTSP data, and the circular byte buffer saves read-ahead RTP media data, if any. However, since data are carefully packetized to fit packet boundary, these two buffers are rarely needed.

The client asks SessionFactory for the found sessions, and can select one of the L2CAP sessions to connect to. As session is established, RTSP_RTP_L2CapConn opens the L2CAP connection and creates the reader/writer objects to act on the connection.

Packetization is required for sending RTSP request. Before an RTSP request is sent out, RTSPBluetoothWriter checks to see if the string exceeds the packet size. If it does not, an RTSP_REQ_MSG header is added to form a message. If the string is longer than the packet size allows, it is broken into multiple messages.

When an RTSP response is expected, RTSPBluetoothReader reads one or more L2CAP messages until a pair of CR/LF is seen. For each message, it peels off the header and searches the data for CR/LF. If there is more data after the CR/LF, it appends the extra data to the read-ahead CircularCharBuffer. Media data is retrieved in a similar manner, RTSP_RTP_L2CapConn reads and checks for RTP_MSG_MARKER message type. It then peels off the header and appends the media data to the media stream, to be return to the media player.

6.5.4 BLUETOOTH PROXY USING L2CAP [13]

Media streaming over Bluetooth requires the client-side library, discussed in Section 6.5.3.2, and the Bluetooth streaming proxy. The proxy bridges the communication between the remote streaming server and the streaming client. The Bluetooth streaming proxy is implemented in C++ using Widcomm Bluetooth SDK [16] and a generic brand USB Bluetooth dongle. The Widcomm Bluetooth driver also needs to be installed for the

dongle to work with the SDK. The Widcomm SDK comes with documentation, header files, and DLL to be incorporated into the project.

The proxy implementation provides a window-based interface for starting and stopping the service. Messages are written to the main windows for debugging. A hierarchy of RTSPRequest, RTSPResponse, RTSPReader, RTSPWriter, and RTPChannel are required. They are similar to those in the Java shared module, and will not be discussed.

Two Bluetooth proxy implementations are provided using L2CAP and RFCOMM. RFCOMM emulates the serial connection, and is built on top of the packet-based L2CAP. The Bluetooth proxy application is designed as a framework to support future Bluetooth proxy implementations. Using this framework, RFCOMM implementation was easily created. Although the Bluetooth proxy implements both L2CAP and RFCOMM proxy services, the Bluetooth client library implements only the L2CAP service. Thus, RFCOMM implementation is currently not used in this project, and is provided only to demonstrate the extensibility of the Bluetooth proxy framework.

A ProxyManager class is created to represent a service. L2CapProxyMgr and RfCommProxyMgr derive from ProxyManager and provide the specific implementations. ProxyManager contains a DataSource object, which serves as the media caching manager.

A FileSystemDataSource is provided to cache media data in file system. It uses a combination of the media URL and media part name to map to a media file in local storage. It maintains this mapping using an STL map.

A base class called BluetoothProxy defines the APIs for streaming proxy and declares methods for handling RTSP requests. Derived proxy classes supply the implementation for reading the Bluetooth messages, and hand the request to BluetoothProxy for handling. BluetoothProxy works with RTSPRequest and RTSPResponse objects only, and does not know about L2CAP or RFCOMM message. BluetoothProxy handles communication with the streaming server on the client's behalf. It uses the socket implementations of RTSPReader and RTSPWriter to send requests to and receive responses from the streaming server.

An L2CAP server can support up to seven clients. Thus L2CapProxyMgr creates seven L2CapProxy objects on initialization. Each L2CapProxy serves on client and handles message packetization and de-packetization. Once all the messages consisting of a request are received, an RTSPRequest object is created and handed to the parent class BluetoothProxy. L2CAP messages are instances of BtMsg.

The Bluetooth proxy communicates with the Bluetooth client on one side, and with the streaming server on the other side on the client's behalf. Proxies keep only the media data, and do not cache SDP data.

When a client connects to the proxy and sends the DESCRIBE request, L2CapProxy immediately creates the socket connections and connect them to the remote server. It then reads the response with SDP data and forwards it back to the client via the Bluetooth connection. Client next sends the SETUP request, which passes through the proxy to the server. The server sets up the session and returns the SETUP response with the session id. L2CapProxy again forwards the response and session id back to the client.

The client can now send a PLAY request. The proxy constructs a key from the media URL and media part file name and looks up its cache, the DataSource object, to see if the requested media file exists. If it finds one, it sends the media data back to the client without consulting the streaming server. If the request media clip is not found, the proxy forwards the PLAY request to the server, receives and forwards back the PLAY response. The server starts sending media data on the RTP connection. The proxy reads media data and forwards to the client, and at the same time saves the data in its cache, via the DataSource object.

Figure 37 shows the sequence diagram illustrating the client-proxy-server interaction.

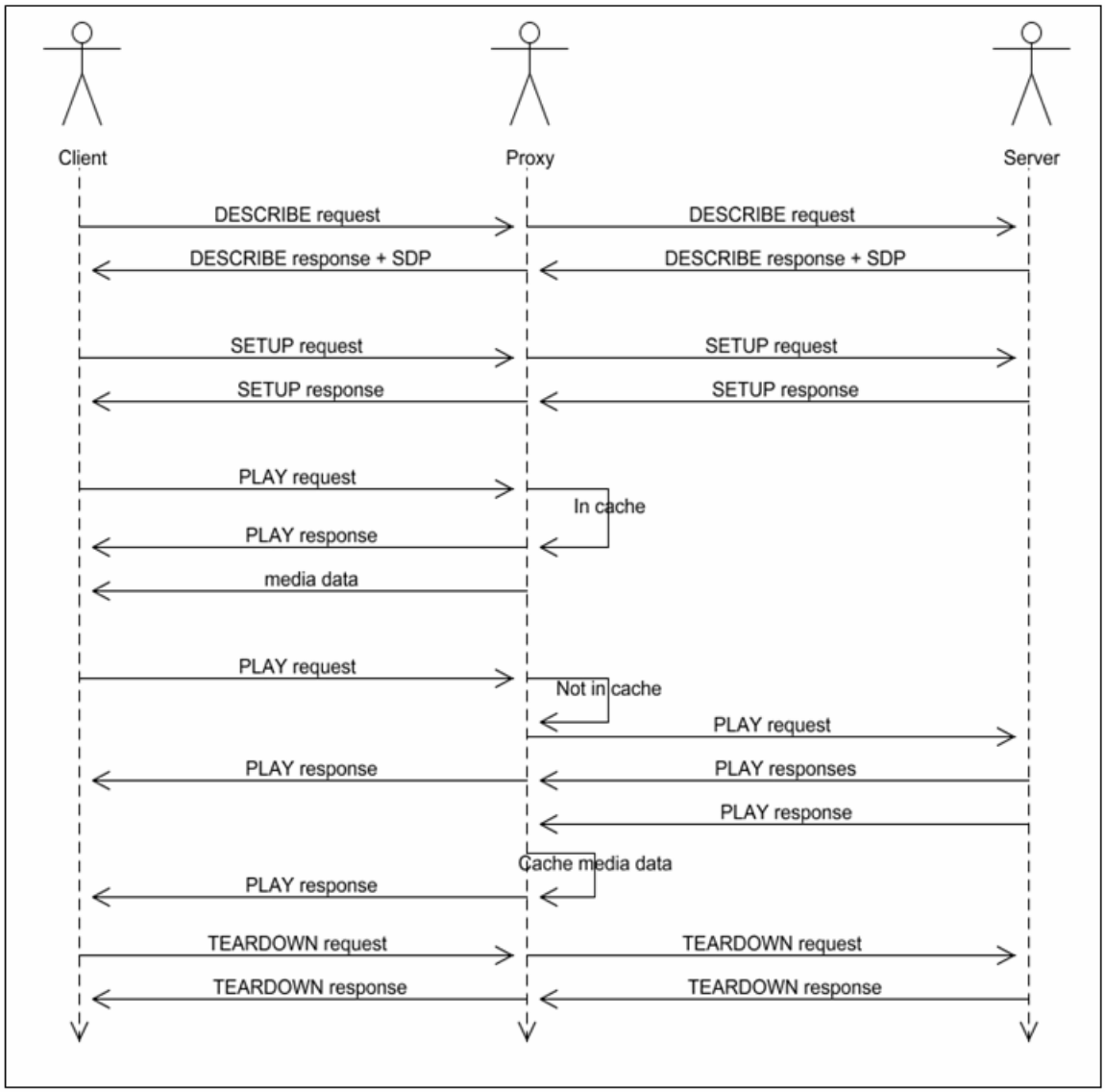


Figure 37. Streaming over Bluetooth activity diagram

6.5.4.1 PACKETIZATION AND TRANSMISSION

Packetization and de-packetization are performed on the proxy as well as on the Bluetooth client. Although L2CAP specification supports up to 64KB packet size, the Bluetooth hardware in reality would not support that much, and would break the packet into multiple smaller packets at the base-band level and reassemble them on the other side. Although longer packets are more efficient, but they also are more susceptible to transmission errors and thus, the default packet size of 672-byte is chosen to avoid complexity.

After the L2CAP connection is established between the client and the proxy, both the proxy and the client determine the default Maximum Transmission Unit (MTU) the connection supports. The default MTU is saved for use later in packet construction.

Before an RTSP request or response is sent, it is serialized into a string. The length of the string is then compared to the MTU, taking into account the message header length (MsgHdrLen). If it is shorter than or equal to (MTU – MsgHdrLen), the message type RTSP_REQ_MSG or RTSP_RESP_MSG is inserted in byte zero, followed by a two-byte message length, and finally the string appended at the end. The message is then sent of in a single L2CAP packet. If the string is longer than (MTU – MsgHdrLen), the message is broken into packets of length (MTU – MsgHdrLen), with RTSP_REQ_MSG or RTSP_RESP_MSG as the message type for the first packet, and CONTINUATION for the following packets.

Similarly, when a packet is received, the message type in its header is inspected. If the message length field following the type is less than or equal to (MTU – MsgHdrLen), the packet contains the complete message, and an RTSP request or response is constructed. If the message type is CONTINUATION, the data in the packet is accumulated until enough has been read, and the accumulated data is used to construct the request/response.

RTP packets containing media data works slightly differently. Since the client already knows the length of the requested clip, it keeps reading in packets of type RTP_MSG_MARKER until enough has been received.

SDP-c data works in the same way. Since the DESCRIBE RTSP response contains the header field *"Content-length"*, which indicates the length of the data following the pair of CR/LF, the SDP-c data can be transferred without using the length field. SDP-c data most likely will be contained in CONTINUATION messages.

CONTINUATION packets always follow a packet that already contains the length for the entire message, and thus do not need to carry a length field. RTP packets, although do not depend on any other packet, do not need a length field either. This is because the length of RTP data is already known in advance by the requester.

The following figures show the format for RTSP request, response, continuation, and RTP packet formats. Notice that, RTSP_REQ_MSG and RTSP_RESP_MSG packets will either contain the whole message, or contain the first part of a multi-packet message.

Figure 38. RTSP request message format

Figure 39. RTSP response message format

CONTINUATION message always follows either an RTSP_REQ_MSG or RTSP_RESP_MSG message and is used only in multi-packet messages.

Figure 40. Continuation message format

RTP_MSG_MARKER packets are always sent independently.

Figure 41. RTP message format

Widcomm SDK [16] provides a framework of C++ classes for handling Bluetooth connection. The L2CapProxy extends the L2CapConn to handle incoming connection request, sending and receiving L2CAP packets. It also has a method called *OnCongestionStatus(BOOL is_congested)*, which is called when the connection is congested or decongested. Experimentation was performed with this method in hope of controlling the packet transmission interval, since the client was experiencing packet loss. However, this method does not work as intended. It only informs the proxy that the connection is congested once, and never notifies again. When congestion is detected for the first time, the proxy stops sending packets, and waits for a notification of channel

decongestion. This event is never received from the framework and the proxy waits forever.

To overcome the SDK traffic control issue, we need to introduce a sleep in between consecutive packet transmissions. There is no rule for the length of this sleep, and we have to go through many trials to find a value that is just long enough to avoid packet loss. The client side also has to wait between packets using the same sleep interval.

The following is a snapshot of the proxy serving a client. From the log, we can see all the RTSP requests and responses, as well as L2CAP packets carrying RTP data to the client. It also indicates that the proxy is using its cache instead of requesting for data from the streaming server.

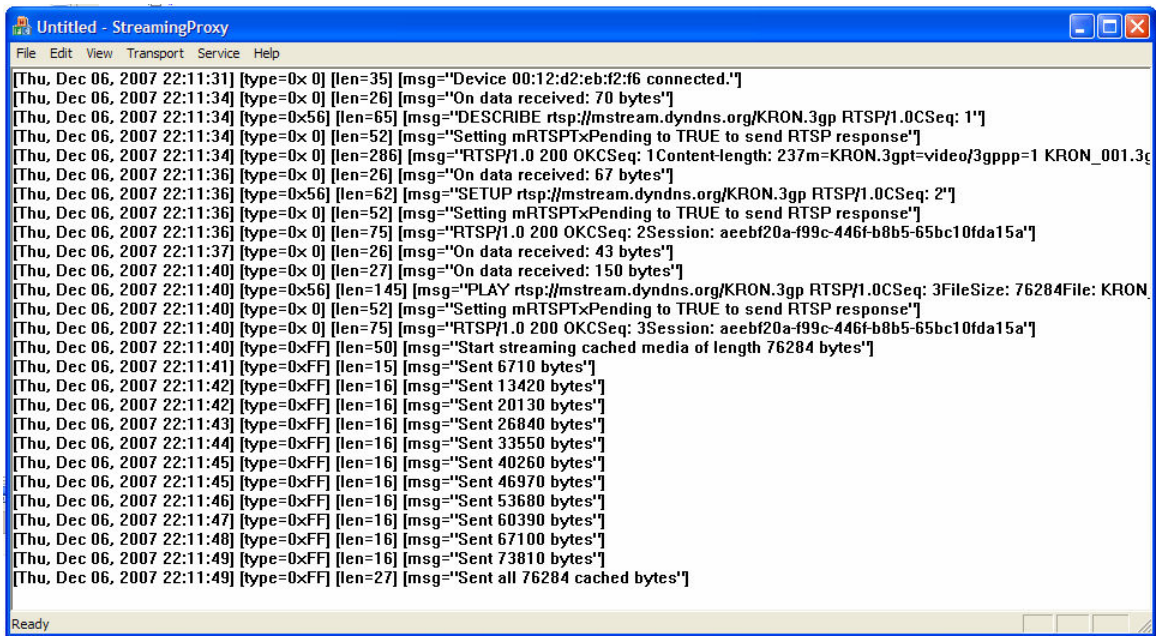


Figure 42. Bluetooth streaming proxy in action

6.5.4.2 DATA CACHING

Caching is done on a media-sub-clip basis by the class FileSystemDataSource. A mapping table is maintained on the proxy to map *(<media-url>|<clip-file-name>)* → *(<clip-file-location)*. The media URL and clip file name are joined together using the character "|" and acts as the key to index to the full path of the media clip. The *DataSource* interface facilitates ease of extension to implement media data caching, i.e., an implementation using a database can be easily implemented.

The following picture shows the mapping table format used by FileSystemDataSource.

```
rtsp://mstream.dyndns.org/KRON.3gp|KRON_001.3gp    ./Media/KRON_001.3gp
rtsp://mstream.dyndns.org/KRON.3gp|KRON_002.3gp    ./Media/KRON_002.3gp
rtsp://mstream.dyndns.org/KRON.3gp|KRON_003.3gp    ./Media/KRON_003.3gp
rtsp://mstream.dyndns.org/KRON.3gp|KRON_004.3gp    ./Media/KRON_004.3gp
rtsp://mstream.dyndns.org/KRON.3gp|KRON_005.3gp    ./Media/KRON_005.3gp
rtsp://mstream.dyndns.org/KRON.3gp|KRON_006.3gp    ./Media/KRON_006.3gp
```

Figure 43. Media cache descriptor format

The full media-URL is used so a new request media URL can readily be used, together with the media part name, to construct the key. This key will be mapped to the full path where the cached media data can be retrieved. Here the cached media files are relative to a data source root directory.

Although Bluetooth transfer is a lot slower than the Internet, caching can greatly improve user experience by avoiding delay, and can especially reduce the load on the streaming server. The user of a Bluetooth proxy is particularly important where a local area network is closed to the public for security reason (so it is not possible for mobile devices to join the network), or for devices that do not have WIFI support. In either case, the network security can be maintained and users can still enjoy the benefit of viewing multimedia without having to pay for data usage.

6.6 STREAMING CLIENT SAMPLE APPLICATION

The client video application is implemented using J2ME. It resides in the packet *com.mstream.client*, consisting of six java files:

- VideoScreen: an interface to be extended by VideoCanvas and VideoForm. This represents the display area where the video will be rendered in.
- VideoCanvas: the canvas-based implementation of VideoScreen. Canvas is "a base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the display" [19], according to the MIDP documentation. To learn more about Canvas, please consult the J2ME MIDP documentation, which can be downloaded from Sun website.

- VideoForm: the form-based implementation of VideoScreen. Form is the high-level display in MIDP.
- SessionChooser: implements a list to display the available session. The list of sessions contains, at the very least, a socket implementation, and Bluetooth services detected in the surroundings. User can choose one of the available session objects for the streaming session.
- SessionChooserListener: a simple callback interface used by the main application, StreamingClient, to detect which session is selected. StreamingClient implements this interface, and registers itself with the SessionChooser object, to be notified when a session is chosen.
- StreamingClient: the client application class. It extends from the MIDP class *javax.microedition.midlet.MIDlet*, and encapsulates the MediaPlayer object, the VideoScreen display object, and handles user interactions. The user can choose a menu item to trigger an action, such as to initialize the streaming session (INIT), describe the media (DESCRIBE), set up the session (SETUP), play (PLAY) and end the session (TEARDOWN).

The following image shows the streaming client playing a sample video. The demo is run on the emulator, using the socket implementation.

Figure 44. Streaming client in emulator

With the original video is split into smaller clips, multiple media players have to be created, with each handling a sub-clip. Since media player preparation is a time-consuming and resource-intensive process, there is a delay between one play and the next. Currently, this is greatly influenced by the device capability. On the emulator, this gap is considerably small, although a screen-switching is noticeable. On the Sony Ericsson S500i, the switching effect is somewhat smoother than the emulator. The Nokia

N80 delivers the worst result. The delay on this device is so significant that a white screen is experienced in between the clips.

7 CONCLUSION

Multimedia production and streaming is a tough process that is highly dependent on the hardware capability and complex software implementation, even on the powerful PC platform. To develop multimedia application on mobile devices, the developers constantly have to deal with device capability and limitation. For mobile devices running on the Symbian platform, streaming has been implemented successfully at the software level without relying on native support. This is because Symbian uses C++ and allows the developers to work closer to the hardware layer. For devices running Java platform, the Java virtual machine only exposes a very high-level framework and completely shields the hardware platform from the developers. Developers have no choice but to stick to a limited set of predefined features.

This project provides a good opportunity to learn the RTSP and RTP specifications, especially the RTP and RTCP packet formats. In attempting to do streaming over TCP/UDP and TCP-Interleaving, a number of RTP and RTCP packet structures have been examined, studied, and implemented. Unfortunately, the current mobile devices do not support playback of partial data, nor do they allow developers to feed media data directly to the hardware, as on the Symbian platform. Using the final approach, multi-subclip, the following goals have been achieved:

- Devising a method for viewing a video by breaking it up into smaller clips, delivering them and displaying them in sequential order, to simulate the streaming effect. This method also requires several custom protocols to be developed: the SDP-c format for describing the media, the RTSP-c protocol for controlling the streaming session, and a custom RTP-channel for media delivery.
- Implementing a custom streaming server that support the request and delivery of video clips, mimicking the RTSP/RTP streaming specification. The server is a multi-threaded service that can serve multiple clients concurrently.

- Implementing a client streaming library that queries the streaming server for media meta-data, initiates the delivery of sub-clips on demand and in advance to minimize delay, and handles media presentation. The library is modularized to support framework extension. Two implementations are provided in the library: using sockets and using Bluetooth technology over a proxy.
- Implementing a Bluetooth proxy service on a LAN-connected PC, that can:
    - Interact with a client via Bluetooth L2CAP.
    - Handle a streaming client session initiation and streaming requests.
    - Set up socket connections to the remote streaming server and relay requests/responses between server and client.
    - Serve requested media data to client from its cache. If the request media is not in the cache, it sends the request to the server, relays the response, receives the media data, forwards media data to client, and saves media data in its cache for future requests.
- Designing and implementing the streaming client library and proxy as frameworks to support future extensions in an easy manner. The proxy framework has been successfully extended using Bluetooth RFCOMM.

This work poses great challenges working with small footprint devices, such as:

- Careful memory management, even in the presence of the Java garbage collector (GC). There is no guarantee about the operation of the GC, and thus much effort has been put into memory utilization as well as Java object reuse.
- Limitations of the device, such as its multimedia capability, low Bluetooth transfer rate and high packet loss rate. Care must be taken in timing and in the delay between packet transmissions.
- No debugging capability. Logging tools have to be developed independently and the logs have to be sent to a PC for inspection.

Despite the clip-switching effect, the video plays fairly well on the Sony Ericsson phone, making the project a good opportunity for studying and experiencing with multimedia on mobile devices, as well as in framework design and development.

8 POTENTIAL FUTURE WORK

If future mobile devices improve in multiplayer support and reduce the delay between players, some research can be put into making the clip transition smoother. The Sony Ericsson devices support progressive download [18], which allows incomplete data to be written to a file on the device and the file is fed to the player. The content handling protocol can continue to write to the file while it is being read by the player.

9 REFERENCES

[1] Video Streaming: Concepts, Algorithms, and Systems. John G. Apostolopoulos, Wai-tian Tan, Susie J. Wee. Mobile and Media Systems Laboratory. HP Laboratories Palo Alto. September 18th, 2002.

[2] A Video Gateway to Support Video Streaming to Mobile Clients. Jens Meggers, Thomas Strang, Anthony Sang-Bum Park.

[3] Streaming in Mobile Networks. MediaLab. August 2004.

[4] SDP: Session Description Protocol. Handley, Jacobson, Perkins. July 2006.

[5] Real Time Streaming Protocol (RTSP). Schulzrinne, Rao, Lanphier. IETF Internet-Draft. February 2nd, 1998.

[6] RTP: A Transport Protocol for Real-Time Applications. Schulzrinne, Casner, Frederick, Jacobson. IETF Internet-Draft. July 2003.

[7] Using RTSP with Firewalls, Proxies, and Other Intermediary Network Devices. RealNetworks. 1998.

[8] Tunneling RTSP/RTP/RTCP in HTTP. Jones, Gentric. IETF Internet-Draft.

[9] QuickTime Streaming Server Modules Programming Guide: Tunneling RTSP and RTP Over HTTP. Apple. 2007.

[10] Experiments in Streaming Content in JavaME. Vikram Goyal. http://today.java.net/lpt/a/314. August 22nd, 2006.

[11] Mobile Media API Version 1.0. Java 2 Platform, Micro Edition. Sun Microsystems. June, 2002.

[12] MIDP: Bluetooth API Developer's Guide. Forum Nokia. October 31st, 2006.

[13] Bluetooth programming for Linux. Marcel Holtmann, Andreas Vedral. Wirelesss Technologies Congress. 2003 Germany.

[14] Games over Bluetooth: Recommendations to Game Developers. Forum Nokia. November 13th, 2003.

[15] An Introduction to Programming the Java APIs for Bluetooth Wireless Technology (JSR 82) on Symbian OS. Martin de Jode. April 5th, 2004.

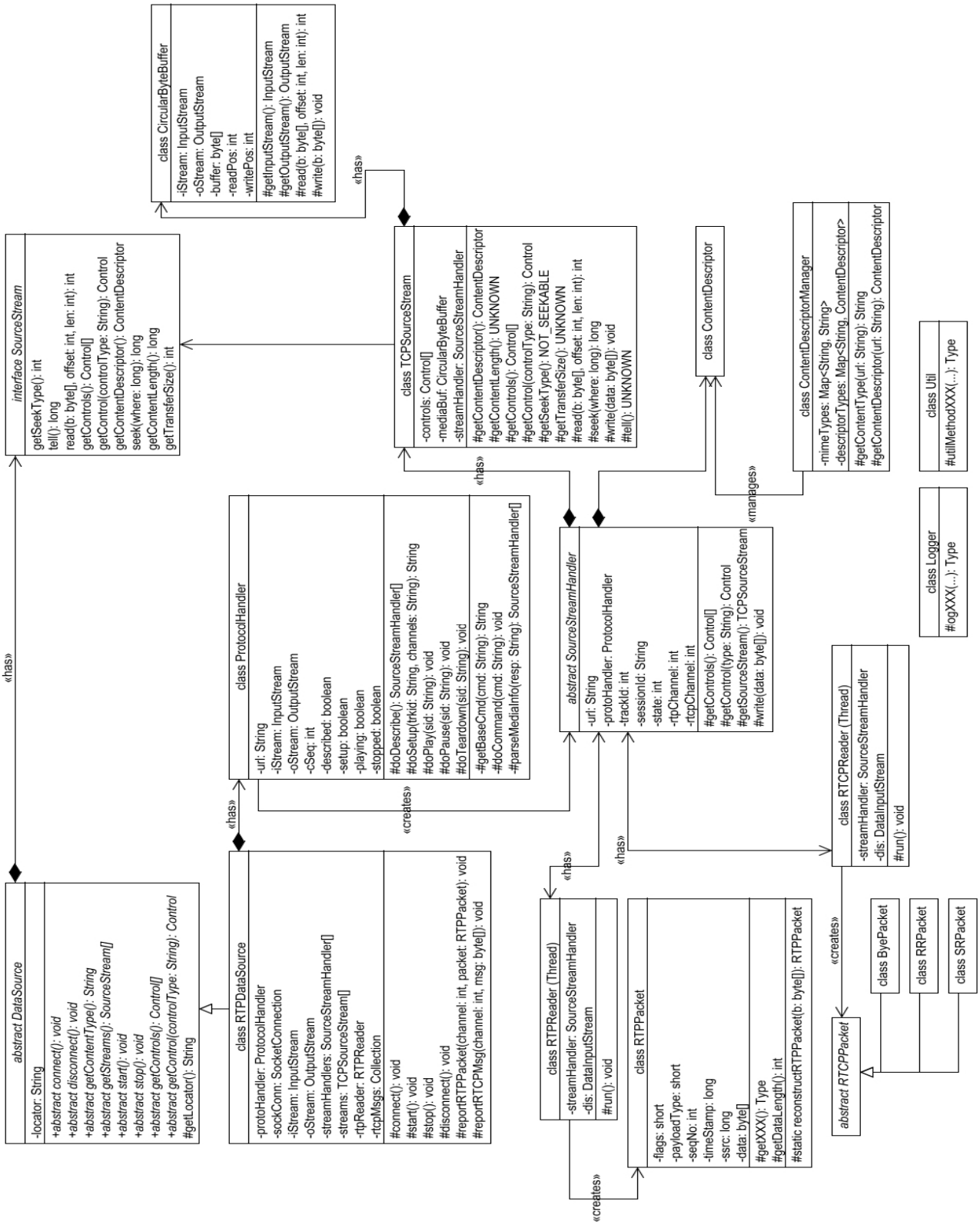
[16] BTW SDK Programmer's Guide. BCM1000-BTW. Broadcom. November 17th, 2006.

[17] Streaming vs. Downloading Video: Understanding the Differences. http://www.streamingmedia.com/article.asp?id=8456&page=1. streamingmedia.com. July 22nd, 2003.

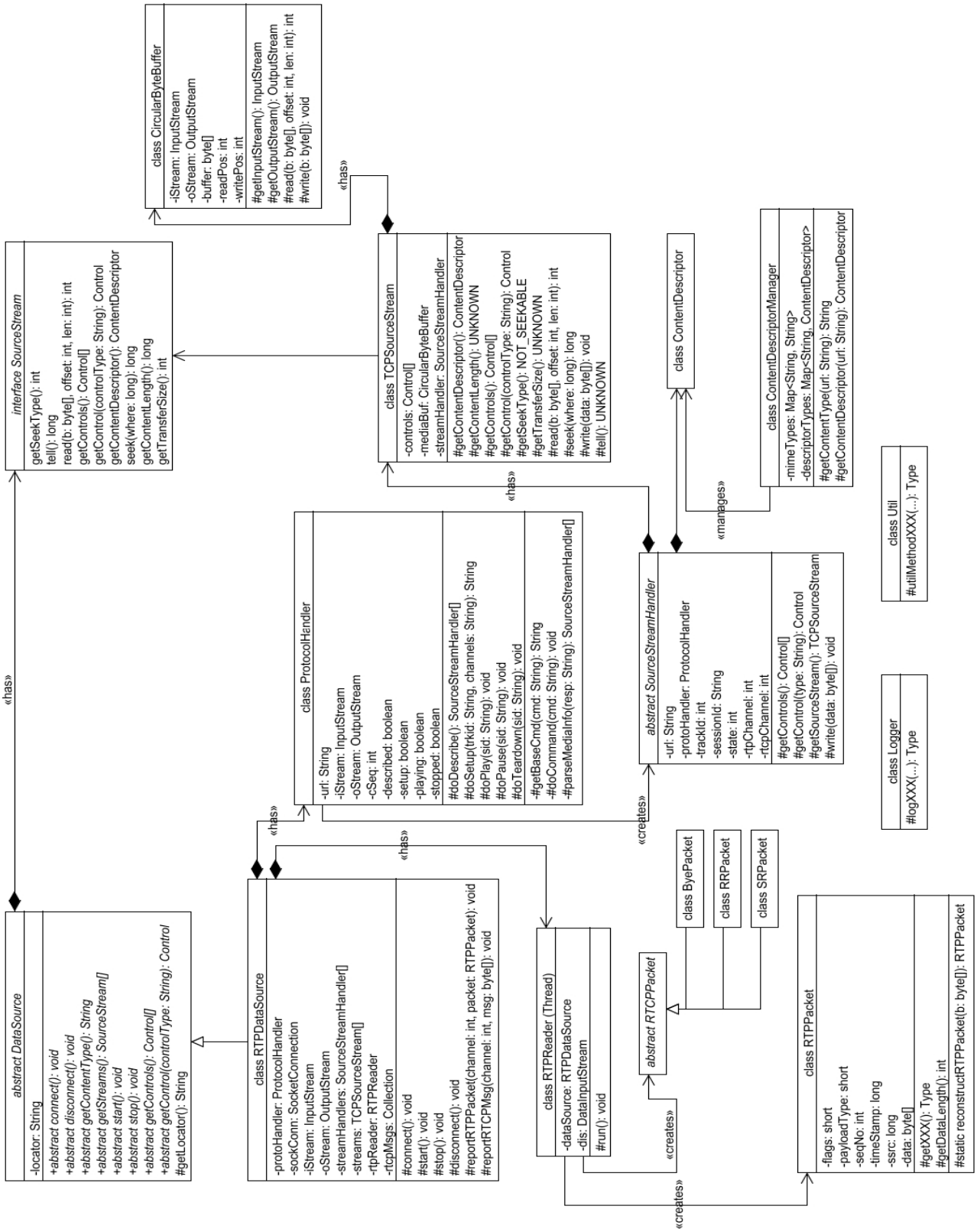
[18] New features in the Mobile Media API (JSR 135): progressive download and audio capture. http://developer.sonyericsson.com/site/global/techsupport/tipstrickscode/java/p_new_features_mobilemedia_api_jsr135.jsp. November 2005.

[19] Mobile Information Device Profile (MIDP 2.0). Sun Microsystems.

Appendix A: *Detailed class diagram for TCP/UPD approach*



Appendix B: Detailed class diagram for TCP-interleaving approach



Appendix C: RTP packet captured from the three tests.

Column	#1	#2	#3		#4	#5		#6	#7
Test case	Native RTSP on W850i				TCP-Interleaved on Emulator			TCP-Interleaved on W850i	
Row	UDP Length	RTP Length	Payload		RTP Length	Payload		RTP Length	Payload
1	306	298	286		298	286		298	286
2	40	32	20		32	20		32	20
3	40	32	20		32	20		32	20
4	40	32	20		32	20		32	20
5	1458	1450	1438		1450	1438		1450	1438
6	469	461	449		461	449		461	449
7	426	418	406		418	406		418	406
8	389	381	369		381	369		381	369
9	391	383	371		383	371		383	371
10	377	369	357		369	357		369	357
11	715	707	695		707	695		707	695
12	709	701	689		701	689		701	689
13	725	717	705		717	705		717	705
14	701	693	681		693	681		693	681
15	613	605	593		605	593		605	593
16	398	390	378		390	378		390	378
17	723	715	703		715	703		715	703
18	717	709	697		709	697		709	697
19	517	509	497		509	497		509	497
20	427	419	407		419	407		419	407
21	307	299	287		299	287		299	287
22	453	445	433		445	433		445	433
23	475	467	455		467	455		467	455
24	453	445	433		445	433		445	433
25	428	420	408		420	408		420	408
26	416	408	396		408	396		408	396
27	322	314	302		314	302		314	302
28	710	702	690		702	690		702	690
29	695	687	675		687	675		687	675
30	1458	1450	1438		1450	1438		1450	1438
31	1458	1450	1438		1450	1438		1450	1438
32	1264	1256	1244		1256	1244		1256	1244
33	346	338	326		338	326		338	326
34	475	467	455		467	455		467	455
35	517	509	497		509	497		509	497
36	341	333	321		333	321		333	321
37	549	541	529		541	529		541	529
38	583	575	563		575	563		575	563
39	596	588	576		588	576		588	576
40	678	670	658		670	658		670	658

41	696	688	676		688	676		688	676
42	595	587	575		587	575		587	575
43	609	601	589		601	589		601	589
44	719	711	699		711	699		711	699
45	737	729	717		729	717		729	717
46	724	716	704		716	704		716	704
47	731	723	711		723	711		723	711
48	733	725	713		725	713		725	713
49	737	729	717		729	717		729	717
50	735	727	715		727	715		727	715
51	741	733	721		733	721		733	721
52	710	702	690		702	690		702	690
53	705	697	685		697	685		697	685
54	701	693	681		693	681		693	681
55	756	748	736		748	736		748	736
56	1458	1450	1438		1450	1438		1450	1438
57	1458	1450	1438		1450	1438		1450	1438
58	1069	1061	1049		1061	1049		1061	1049
59	654	646	634		646	634		646	634
60	666	658	646		658	646		658	646
61	637	629	617		629	617		629	617
62	697	689	677		689	677		689	677
63	719	711	699		711	699		711	699
64	680	672	660		672	660		672	660
65	754	746	734		746	734		746	734
66	793	785	773		785	773		785	773
67	756	748	736		748	736		748	736
68	832	824	812		824	812		824	812
69	965	957	945		957	945		957	945
70	844	836	824		836	824		836	824
71	978	970	958		970	958		970	958
72	1058	1050	1038		1050	1038		1050	1038
73	997	989	977		989	977		989	977
74	1245	1237	1225		1237	1225		1237	1225
75	1308	1300	1288		1300	1288		1300	1288
76	1114	1106	1094		1106	1094		1106	1094
77	1360	1352	1340		1352	1340		1352	1340
78	1390	1382	1370		1382	1370		1382	1370
79	1180	1172	1160		1172	1160		1172	1160
80	1145	1137	1125		1137	1125		1137	1125
81	924	916	904		916	904		916	904
82	1458	1450	1438		1450	1438		1450	1438
83	1458	1450	1438		1450	1438		1450	1438
84	274	266	254		266	254		266	254
85	829	821	809		821	809		821	809
86	757	749	737		749	737		749	737
87	715	707	695		707	695		707	695
88	759	751	739		751	739		751	739

89	677	669	657		669	657		669	657
90	617	609	597		609	597		609	597
91	586	578	566		578	566		578	566
92	534	526	514		526	514		526	514
93	493	485	473		485	473		485	473
94	449	441	429		441	429		441	429
95	415	407	395		407	395		407	395
96	379	371	359		371	359		371	359
Total	69415	68647	67495		68647	67495		68647	67495